GemStone®

GemBuilder for Smalltalk User's Guide

For use with Cincom Smalltalk VisualWorks

Version 7.3

January 2010

GEMSTONE S™

INTELLECTUAL PROPERTY OWNERSHIP

This documentation is furnished for informational use only and is subject to change without notice. GemStone Systems, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this documentation.

This documentation, or any part of it, may not be reproduced, displayed, photocopied, transmitted, or otherwise copied in any form or by any means now known or later developed, such as electronic, optical, or mechanical means, without express written authorization from GemStone Systems, Inc.

Warning: This computer program and its documentation are protected by copyright law and international treaties. Any unauthorized copying or distribution of this program, its documentation, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted under the maximum extent possible under the law.

The software installed in accordance with this documentation is copyrighted and licensed by GemStone Systems, Inc. under separate license agreement. This software may only be used pursuant to the terms and conditions of such license agreement. Any other use may be a violation of law.

Use, duplication, or disclosure by the Government is subject to restrictions set forth in the Commercial Software - Restricted Rights clause at 52.227-19 of the Federal Acquisitions Regulations (48 CFR 52.227-19) except that the government agency shall not have the right to disclose this software to support service contractors or their subcontractors without the prior written consent of GemStone Systems, Inc.

This software is provided by GemStone Systems, Inc. and contributors “as is” and any expressed or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall GemStone Systems, Inc. or any contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

COPYRIGHTS

This software product, its documentation, and its user interface © 1986-2010 GemStone Systems, Inc. All rights reserved by GemStone Systems, Inc.

PATENTS

GemStone is covered by U.S. Patent Number 6,256,637 “Transactional virtual machine architecture”, Patent Number 6,360,219 “Object queues with concurrent updating”, and Patent Number 6,567,905 “Generational Garbage Collector”. GemStone may also be covered by one or more pending United States patent applications.

TRADEMARKS

GemStone, **GemBuilder**, **GemConnect**, and the GemStone logos are trademarks or registered trademarks of GemStone Systems, Inc. in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Sun, **Sun Microsystems**, **Solaris**, and **SunOS** are trademarks or registered trademarks of Sun Microsystems, Inc. All **SPARC** trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. **SPARCstation** is licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

HP and **HP-UX** are registered trademarks of Hewlett Packard Company.

Intel and **Pentium** are registered trademarks of Intel Corporation in the United States and other countries.

Microsoft, **MS**, **Windows**, **Windows 2000** and **Windows XP** are registered trademarks of Microsoft Corporation in the United States and other countries.

Linux is a registered trademark of Linus Torvalds and others.

Red Hat and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc. in the United States and other countries.

AIX and **POWER4** are trademarks or registered trademarks of International Business Machines Corporation.

Other company or product names mentioned herein may be trademarks or registered trademarks of their respective owners. Trademark specifications are subject to change without notice. All terms mentioned in this documentation that are known to be trademarks or service marks have been appropriately capitalized to the best of our knowledge; however, GemStone cannot attest to the accuracy of all trademark information. Use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.

GemStone Systems, Inc.
1260 NW Waterhouse Avenue, Suite 200
Beaverton, OR 97006

Preface

About This Manual

This manual describes GemBuilder for Smalltalk®, an environment for developing Gemstone applications using VisualWorks Smalltalk.

GemBuilder for Smalltalk consists of two parts: a programming interface between your client Smalltalk application code and the GemStone object repository, and a GemStone programming environment.

The programming interface provides facilities for managing the relationship between objects on the GemStone server and in client Smalltalk, allowing objects to be available on the client and updated on the shared GemStone server.

The GemBuilder programming environment provides a set of integrated tools for programming in GemStone's version of Smalltalk.

GemBuilder supports both GemStone/S and GemStone/S 64 Bit. There are minor differences between the features available when using different server products. Please consult the documentation for the server product and version you are using for specific details on that product.

Prerequisites

To make use of the information in this manual, you need to be familiar with the GemStone object server and with GemStone's Smalltalk programming language as described in the *GemStone/S Programming Guide*. That book explains the basic concepts behind the language and describes the most important GemStone kernel classes.

In addition, you should be familiar with the VisualWorks Smalltalk language and programming environment as described in the VisualWorks Smalltalk product manuals.

Finally, this manual assumes that the GemStone system has been correctly installed on your host computer as described in the *GemStone/S System Administration Guide* and that your system meets the requirements listed in your *GemBuilder for Smalltalk Installation Guide*.

How This Manual is Organized

Chapter 1, Basic Concepts, describes the overall design of a GemBuilder application and presents the fundamental concepts required to understand the interface between client Smalltalk and the GemStone object server.

Chapter 2, Communicating with the GemStone Object Server, explains how to communicate with the GemStone object server by initiating and managing GemStone sessions.

Chapter 3, Using the GemStone Programming Tools, explains how to use the GemStone browsers and tools to create classes and methods in GemStone and to debug GemStone Smalltalk code.

Chapter 4, Debugging, explains breakpoints and other ways you can use GemBuilder tools to debug your application.

Chapter 5, Sharing Objects, describes the various mechanisms GemBuilder can use to coordinate your application's local objects with objects in the GemStone object server, thus making them persistent and sharable.

Chapter 6, Connectors, explains how to connect your application's local objects to objects in the GemStone repository in order to implement object sharing and allow your application to manipulate objects in the server.

Chapter 7, Managing Transactions, discusses the process of committing a transaction, the kinds of conflicts that can prevent a successful commit, and how to avoid or resolve such conflicts.

Chapter 8, Security and Object Access, describes the security mechanisms that are available in GemBuilder and explains how to control access to objects in a multiuser environment. It explains how to use the GemBuilder tools to manage access to objects and administer user accounts.

Chapter 9, Exception Handling, discusses errors: how to handle them and how to recover from them

Chapter 10, Schema Modification and Coordination, explains how GemStone supports schema modification by maintaining versions of classes in class histories. It describes the Class Version Browser and explains how to use it. It also explains how to synchronize schema modifications between the client and GemStone.

Chapter 11, Performance Tuning, discusses ways that you can tune your application to optimize performance and minimize maintenance overhead. It describes the configuration parameters available for tuning a GemBuilder application, and it explains how to configure GemBuilder for Smalltalk to optimize your application's performance.

Appendix A, GemBuilder Configuration Parameters, describes the GemBuilder for Smalltalk configuration options and how to set and use them.

Appendix B, Packaging Runtime Applications, provides brief instructions for packaging runtime applications.

Appendix C, Client Smalltalk and GemStone Smalltalk, outlines a few general and syntactical differences between the client Smalltalk and GemStone Smalltalk dialects.

Other Useful Documents

You will find it useful to look at documents that describe other components of the GemStone data management system:

- The *GemStone Programming Guide* describes GemStone Smalltalk and discusses managing common operations.
- The *GemBuilder for Smalltalk Tutorial* provides an introduction to programming GemStone applications with GemBuilder.
- A description of the behavior of each GemStone kernel class is available in the class comments in the GemStone Smalltalk repository.

- In addition, if you will be acting as a system administrator, or developing software for someone else who must play this role, you will need to read the *System Administration Guide* for your GemStone/S server product.

Technical Support

GemStone provides several sources for product information and support. The product-specific manuals provide extensive documentation, and should be your first source of information.

GemStone Web Site: http://support.gemstone.com

GemStone's Technical Support website provides a variety of resources to help you use GemStone products. Use of this site requires an account, but registration is free of charge. To get an account, just complete the Registration Form, found in the same location. You'll be able to access the site as soon as you submit the web form.

The following types of information are provided at this web site:

Documentation for GemBuilder for Smalltalk is provided in PDF format. This is the same documentation that is included with your GemBuilder for Smalltalk product.

Release Notes and **Install Guides** for your product software are provided in PDF format in the Documentation section.

Downloads and **Patches** provide code fixes and enhancements that have been developed after product release, and past and current versions of GemBuilder for Smalltalk.

Bugnotes, in the Learning Center section, identify performance issues or error conditions that you may encounter when using a GemStone product. A bugnote describes the cause of the condition, and, when possible, provides an alternative means of accomplishing the task. In addition, bugnotes identify whether or not a fix is available, either by upgrading to another version of the product, or by applying a patch. Bugnotes are updated regularly.

TechTips, also in the Learning Center section, provide information and instructions for topics that usually relate to more effective or efficient use of GemStone products.

Community Links provide customer forums for discussion of GemStone product issues.

Technical information on the GemStone Web site is reviewed and updated regularly. We recommend that you check this site on a regular basis to obtain the latest technical information for GemStone products.

Help Requests

You may need to contact Technical Support directly for the following reasons:

- Your technical question is not answered in the documentation.
- You receive an error message that directs you to contact GemStone Technical Support.
- You want to report a bug.
- You want to submit a feature request.

Questions concerning product availability, pricing, keyfiles, or future features should be directed to your GemStone account manager.

When contacting GemStone Technical Support, please be prepared to provide the following information:

- Your name, company name, and GemStone/S license number
- The GemStone product and version you are using
- The hardware platform and operating system you are using
- A description of the problem or request
- Exact error message(s) received, if any

Your GemStone support agreement may identify specific individuals who are responsible for submitting all support requests to GemStone. If so, please submit your information through those individuals. All responses will be sent to authorized contacts only.

For non-emergency requests, the support website is the preferred way to contact Technical Support. Only designated support contacts may submit help requests via the support website. If you are a designated support contact for your company, or the designated contacts have changed, please contact us to update the appropriate user accounts.

Website: http://techsupport.gemstone.com

Email: support@gemstone.com

Telephone: (800) 243-4772 or (503) 533-3503

Requests for technical assistance may be submitted online, or by email or by telephone. We recommend you use telephone contact only for more serious requests that require immediate evaluation, such as a production system that is non-operational. In these cases, please also submit your request via the web or email, including pertinent details such error messages and relevant log files.

If you are reporting an emergency by telephone, select the option to transfer your call to the technical support administrator, who will take down your customer information and immediately contact an engineer.

Non-emergency requests received by telephone will be placed in the normal support queue for evaluation and response.

24x7 Emergency Technical Support

GemStone/S offers, at an additional charge, 24x7 emergency technical support. This support entitles customers to contact us 24 hours a day, 7 days a week, 365 days a year, if they encounter problems that cause their production application to go down, or that have the potential to bring their production application down. Contact your GemStone/S account manager for more details.

Contents

Chapter 1. Basic Concepts

1.1 The GemStone Object Server 20
1.2 GemBuilder for Smalltalk 21
The Programming Interface 22
Transparent Access to GemStone 22
GemStone's Smalltalk Language 23
The GemBuilder Tools 24
1.3 Designing a GemStone Application: an Overview 25
Which objects should be stored and shared? 25
Which objects should be secured? 26
Which objects should be connected? 26
How should transactions be handled? 27
How can performance be improved? 27
1.4 Delivery and Deployment 27
Public and Private Classes and Methods 28

Chapter 2. Communicating with the GemStone Object Server

2.1 Client Libraries 32
2.2 GemStone Sessions 32
RPC and Linked Sessions 33
2.3 Session Control in GemBuilder 34
Session Parameters 35
Defining Session Parameters Programmatically 36
2.4 The GemStone Session Browser 38
Starting the Session Browser 38
Opening the Session Parameters Editor 38
Managing Session Parameters 39
2.5 Logging In to and Logging Out of GemStone 40
Programmatic Session Management 41
Logging In to GemStone 41
The Current Session 41
Logging Out of GemStone 42
Session Management Using the Session Browser 43
Logging In to GemStone 43
Setting the Current Session 43
Logging Out of GemStone 43
2.6 Session Dependents 44

Chapter 3. Using the GemStone Programming Tools

3.1 GemStone Menu 50
3.2 Browsing Code 52
Symbol List Pane 54
Class Pane 55
Pop-up Text Pane Menu 56
3.3 The System Workspace 57
3.4 Coding 58
About GemStone Smalltalk Classes 58
Defining a New Class 60
Subclass Creation Methods 62
Private Instance Variables 63
Modifying an Existing Class 63
Defining Methods 64

Public and Private Methods 64
Reserved and Optimized Selectors 65
Saving Class and Method Definitions in Files 65
Handling Errors While Filing In 68

Chapter 4. Debugging

4.1 Debugging Overview. 70
4.2 Inspectors 70
4.3 Breakpoints 73
Breakpoints for Primitive Methods 75
Breakpoints for Optimized Methods 75
The Breakpoint Browser 76
4.4 Debugger 77
Disabling the Debugger 78
4.5 Stack Traces 79

Chapter 5. Sharing Objects

5.1 Which Objects to Share? 82
5.2 Class Mapping 83
Automatic Class Generation and Mapping. 84
Schema Mapping 85
Behavior Mapping 85
Mapping and Class Versions. 86
5.3 Forwarders. 86
Sending Messages 87
Arguments 88
Results 88
Defunct Forwarders. 88
5.4 Replicates 89
Synchronizing State 90
Faulting 90
Flushing. 91
Marking Modified Objects Dirty Automatically 91
Marking Modified Objects Dirty Manually. 92
Minimizing Replication Cost. 92
Instance Variable Mapping 92

Stubbing 95
Replication Specifications 99
Customized Flushing and Faulting 105
Modifying Instance Variables During Faulting 105
Modifying Instance Variables During Flushing 107
Mapping Classes With Different Formats 109
Limits on Replication 109
Replicating Client Smalltalk BlockClosures 109
Block Callbacks 113
Replicating Collections with Instance Variables 114
5.5 Precedence of Replication Mechanisms 114
5.6 Evaluating Smalltalk Code on the GemStone server 115
5.7 Converting Between Forms 117

Chapter 6. Connectors

6.1 Connecting Root Objects. 122
Scope 124
Verifying Connections 125
Initializing 125
Updating Class Definitions. 126
6.2 Connecting and Disconnecting 126
6.3 Kinds of Connectors 127
Connection Order 128
Lookup 128
Connecting by Identity: Fast Connectors 129
6.4 Making and Managing Connectors. 130
Making Connectors Programmatically 130
Creating Connectors 131
Setting the Postconnect Action. 131
Adding Connectors to a Connector List 132
Session Control 133
The Connector Browser 136
The Group Pane 137
The Connector Pane. 137
The Control Panel 138
Postconnect Action 139

Chapter 7. Managing Transactions

7.1 Transaction Management: an Overview 142
7.2 Operating Inside a Transaction 143
Committing a Transaction 144
Aborting a Transaction 145
Avoiding or Handling Commit Failures 145
7.3 Operating Outside a Transaction 146
Being Signaled to Abort 147
7.4 Transaction Modes 148
Automatic Transaction Mode 148
Manual Transaction Mode 149
Choosing Which Mode to Use 149
Switching Between Modes 149
7.5 Managing Concurrent Transactions 150
Read and Write Operations 150
Optimistic and Pessimistic Concurrency Control 151
Setting the Concurrency Mode. 152
Setting Locks. 153
Releasing Locks Upon Aborting or Committing. 155
7.6 Reduced-conflict Classes. 156
7.7 Changed Object Notification 157
7.8 Gem-to-Gem Notification 158
7.9 Asynchronous Event Error Handling. 160

Chapter 8. Security and Object Access

8.1 GemStone Security 162
Login Authorization 162
The UserProfile. 162
Controlling Visibility of Objects with SymbolLists 162
System Privileges 163
Protecting Methods 163
Object-level Security 163
Segments 163
8.2 GemStone Administration Tools 164
The Segment Tool 164
Segment Definition Area 166

Group Definition Area 167
Segment Tool Menus 167
Using the Segment Tool 171
The Symbol List Browser. 174
The Clipboard 175
Symbol List Browser Menus 175
User Account Management Tools. 177
GemStone User List 177
GemStone User Dialog 179
The Privileges Dialog 181

Chapter 9. Exception Handling

9.1 GemStone Errors and Exception Classes. 183
9.2 Handling Exceptions 185
User-defined Errors. 186
9.3 Interrupting GemStone Execution 188

Chapter 10. Schema Modification and Coordination

10.1 Schema Modification 190
Instance Migration Within GemStone 190
Setting the Migration Destination 191
Migrating Objects 191
Things to Watch Out For 192
Instance Variable Mapping in Migration 192
10.2 Schema Coordination 194
10.3 The Class Version Browser. 195
Menus in the Class Version Browser 195

Chapter 11. Performance Tuning

11.1 Profiling 200
Profiling Client Smalltalk Execution 200
Main Statistics 200
Cache Inventory Statistics 203
VSD. 203
11.2 Selecting the Locus of Control 204

Locus of Execution 204
Relative Platform Speeds 204
Cost of Data Management 204
GemStone Optimization 205
11.3 Replication Tuning 205
Controlling the Fault Level. 206
Preventing Transient Stubs. 206
Setting the Traversal Buffer Size. 207
11.4 Optimizing Space Management 207
Explicit Stubbing 207
Using Forwarders 209
11.5 Using Primitives 209
11.6 Multiprocess Applications 209
Blocking and Nonblocking Protocol 210
One Process per Session 210
Multiple Processes per Session 210
Coordinating Transaction Boundaries. 211
Coordinating Flushing 211
Coordinating Faulting. 212
Using the VisualWorks Application Model 212

Appendix A. GemBuilder Configuration Parameters

A.1 Setting Configuration Parameters 215
A.2 GemBuilder Configuration Parameter. 217
alwaysUseGemCursor 219
assertionChecks 219
autoMarkDirty 219
blockingProtocolRpc 219
blockReplicationEnabled 220
blockReplicationPolicy 220
bulkLoad 220
clientMapCapacity. 220
clientMapFinalizerPriority 221
confirm 222
connectorNilling 222
connectVerification 222
defaultFaultPolicy 223
deprecationWarnings 223

eventPollingFrequency 223
eventPriority 224
faultLevelLnk 224
faultLevelRpc 224
forwarderDebugging 225
freeSlotsOnStubbing 225
fullCompression 225
gcedObjBufferSize 226
generateClassConnectors 226
generateClientClasses 226
generateServerClasses 227
InitialDirtyPoolSize 227
libraryName 228
pollForAsynchronousEvents 228
pollForRpcResponse 229
removeInvalidConnectors 229
rpcSocketWaitTimeoutMs 229
serverMapLeafCapacity 230
stubDebugging 230
traversalBufferSize 231
verbose 231
A.3 The Settings Browser 231
Opening the Settings Browser 232
The Settings Browser 232
Parameter Categorization 233

Appendix B. Packaging Runtime Applications

B.1 Prerequisites 235
Names 235
Replicating Blocks 236
Defunct Stubs and Forwarders 236
Shared Libraries 236
B.2 Packaging 236

Appendix C. Client Smalltalk and GemStone Smalltalk

C.1 Language Differences 237
Selection Blocks 238

Array Constructors 238
C.2 TimeZone handling 238

Chapter 1

Basic Concepts

This chapter describes the overall design of a GemBuilder application and presents the fundamental concepts required to understand the interface between client Smalltalk and the GemStone object server.

The GemStone Object Server
introduces GemStone and its architecture and explains the part each component plays in the system.

GemBuilder for Smalltalk
outlines the basic features of GemBuilder that allow you to access GemStone objects from your Smalltalk application, and describes the basic programming functions that GemBuilder provides.

Designing a GemStone Application: an Overview
outlines the basic steps involved in producing a client / server application with GemBuilder.

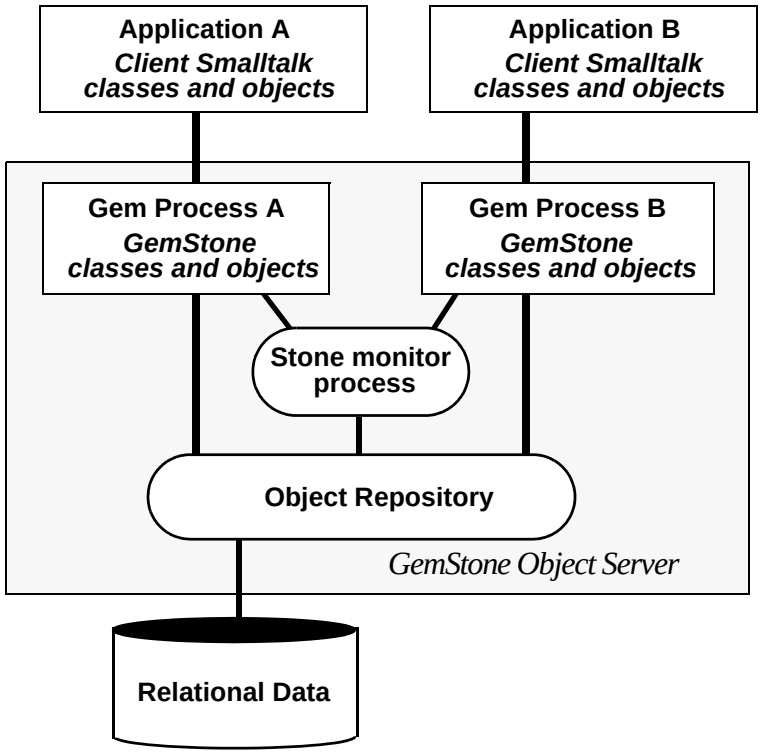
1.1 The GemStone Object Server

The GemStone object server supports multiple concurrent users of a large repository of objects. GemStone provides efficient storage and retrieval of large sets of objects, resiliency to hardware and software failures, protection for object integrity, and a rich set of security mechanisms.

The GemStone object server consists of three main components: a *repository* for storing persistent, shared objects; a monitor process called the *Stone*, and one or more user processes, called *Gems*.

Figure 1.1 shows how the object server supports clients in a Smalltalk application environment.

Figure 1.1 The GemStone Object Server



The *object repository* is a multiuser, disk-based Smalltalk image containing shared application objects and GemStone kernel classes. It is composed of files

(known to GemStone as *extents*) that can reside on a single machine or can be distributed among several networked hosts. The repository can also include GemConnect objects representing data stored in third-party relational databases.

Your Smalltalk application program treats the repository as a single unit, regardless of where its elements physically reside.

A *Gem* is an executable process that your application creates when it begins a GemStone session. A Gem acts as an object server for one session, providing a single-user view of the multiuser GemStone repository. A Gem reads objects from the repository, executes GemStone Smalltalk methods, and updates the repository.

Each Gem represents a single session. An application can create more than one session, each representing an internally-consistent single view of the repository. When a Gem *commits a transaction*, it modifies the shared repository and updates its own view of the repository.

The *Stone* monitor process handles locking and controls concurrent access to objects in the repository, ensuring integrity of the stored objects. Each repository is monitored by a single Stone.

Despite its central role in coordinating the work of all individual Gems, the Stone is surprisingly unintrusive. To optimize throughput for all users, most processing is handled by the Gems, which often interact directly with the repository. The Stone intervenes only when required to ensure the integrity of the multiuser repository.

1.2 GemBuilder for Smalltalk

GemBuilder for Smalltalk is a set of classes and primitives that can be installed in a Smalltalk image. With the functionality provided by GemBuilder, you can:

- store your client Smalltalk application objects in the GemStone server;
- import GemStone objects into client Smalltalk as client Smalltalk objects;
- allow your application objects to be transparently replicated and maintained both in the client and in the server, or allow some objects to reside only in the server but be accessible on the client;
- arrange for messages sent to client Smalltalk objects to be forwarded and executed in the GemStone server by corresponding server objects;

- use GemStone's programming tools to develop GemStone server classes and methods to operate on your application objects; and
- perform certain system functions, such as committing transactions and starting or ending GemStone sessions.

The Programming Interface

Your client Smalltalk application creates a GemStone session by using GemBuilder to log in to GemStone, creating a Gem process to serve your application. Many Gem processes can actively communicate with a single repository at the same time.

As Figure 1.1 illustrates, several applications can work concurrently with a single repository, with each application viewing the repository as its own. GemStone coordinates transactions between each of the applications and the repository.

Transparent Access to GemStone

The interface between your client Smalltalk application and GemStone can be relatively seamless.

Many of the classes in the base client Smalltalk image are mapped to comparable GemStone server classes, and additional class mappings can be created either automatically or explicitly. GemBuilder is also able to automatically generate GemStone server classes from client classes, and vice versa, as necessary. Your client objects can be replicated in the GemStone server, and GemStone server objects can be replicated in client Smalltalk.

The simplest way to make a client Smalltalk object persistent—that is, to store it in GemStone—is to define a *connector* for the object. A *connector* is an object that knows how to resolve a client Smalltalk object and a GemStone server object and how to establish a relationship between them when a session logs into GemStone. Once you've defined a connector for the two objects, GemBuilder maintains the relationship between the shared GemStone server object and the private client Smalltalk object, updating values from the repository to your application and vice versa, as necessary. The connector ensures that if a shared GemStone server object is modified, the application's client Smalltalk counterpart is updated automatically.

Your client Smalltalk application updates shared objects in the repository by sending a `commit` message to a session. With a successful commit, changes to objects in the current session are propagated to the shared object repository in GemStone. Once you have committed a transaction, your application objects are updated with the most recently saved state of the repository, incorporating changes made by other users.

While, for the most part, GemBuilder will automatically manage objects in both the client Smalltalk and in the GemStone server, you can exert as much control as you want over how objects are stored and used. GemBuilder provides tools that let you specify customized policies for translating between your client Smalltalk and GemStone server objects.

Chapter 6 describes GemBuilder's mechanisms for making your client Smalltalk objects persistent, and Chapter 11 explains how to tune the system to minimize maintenance overhead and optimize performance.

GemStone's Smalltalk Language

GemStone provides a version of Smalltalk that supports multiple concurrent users of the shared object repository through such features as session management, reduced-conflict collection classes, querying, transaction management, and persistence.

GemStone Smalltalk is like single-user client Smalltalk in both its organization and syntax. Objects are defined by classes based on common structure and protocol and classes are organized into an *is-a* hierarchy, rooted at class Object. The class hierarchy is extensible; new classes can be added as required to model an application. The behavior of common classes conforms to the ANSI standard for Smalltalk. GemStone's class hierarchy is discussed in the introductory chapter to the *GemStone Programming Guide*.

The most significant difference between GemStone Smalltalk and client Smalltalk lies in GemStone's support for a multiuser environment in which persistent objects can be shared among many users.

As an object server, GemStone must address the same key issues as conventional information storage systems that support multiple concurrent users. For this reason, GemStone's Smalltalk includes classes for:

- managing concurrent access to information,
- protecting information from unauthorized access, and
- keeping stored information secure and restoring it in the event of a failure.

You should be aware of a few differences between GemStone Smalltalk and client Smalltalk in syntax and in the behavior of some of the classes. A summary of these differences can be found in Appendix C.

The GemBuilder Tools

GemBuilder's programming environment provides tools for programming in GemStone Smalltalk. The following tools are described in detail in subsequent chapters of this manual:

- A *GemStone System Browser* lets you examine, modify, and create GemStone classes and methods.
- A *GemStone System Workspace* provides easy access to commonly used GemStone Smalltalk expressions.
- *GemStone Inspectors* let you examine and modify the state of GemStone objects.
- A *GemStone Breakpoint Browser* and a *Debugger* let you examine and dynamically modify the state of a running GemStone application.
- A *Session Browser* allows you to manage sessions and transactions.
- A *Connector Browser* allows you to manage the connectors that establish relationships between client Smalltalk and GemStone server objects.
- A *Class Version Browser* can be used for examining a class history, inspecting instances, migrating instances, deleting versions, and moving versions to another class history.
- A *Symbol List Browser* allows you to examine the GemStone Symbol Lists associated with UserProfiles, add and delete dictionaries from these lists, and manipulate the entries in those dictionaries.
- A *User Account Management Tool* allows you to create new user accounts, change account passwords, and assign group membership.
- A *Segment Tool* facilitates managing GemStone authorization at the object level by controlling how objects are assigned to segments.

1.3 Designing a GemStone Application: an Overview

Many GemStone users start with an application they have already written in Smalltalk. Their mission is to transform that application into one that makes meaningful use of GemStone's features: persistence, multiuser access, security, integrity, and the ability to store and manage large quantities of information.

As a GemStone programmer, your application design and porting efforts involve the following tasks:

- choosing the objects that should be stored and shared,
- deciding which objects need to be secured,
- establishing connections between root objects in the client and the server,
- deciding when to commit transactions and how to handle concurrency control, and
- tuning your application for optimal performance.

This section gives you an overview of these steps and points you to the chapters that discuss these topics in detail.

Which objects should be stored and shared?

Your application will typically have two kinds of objects: those that must persist between sessions and be shared among users, and those that represent a transient state. Your first task is to identify the objects that make up your application and decide which ones need to be stored and shared. Making objects persistent unnecessarily can degrade performance and complicate programming.

Use the GemStone server to store those objects that need to exist between sessions and must be shared with other users. For example, objects representing information in your application such as financial statements, employee health records, or library book cards would certainly require storage in the server. Some methods for manipulating the persistent data can also be usefully coded in GemStone Smalltalk and stored in GemStone for improved efficiency.

You don't need to store transient session objects that no one else will ever need on the server; such objects can remain in the client. For example, suppose you generate a report from the financial statements stored in GemStone. Once you view or print the report it has served its purpose; the next time you need a report you will generate a new one. The report and its component objects can exist simply as client Smalltalk objects; they don't need to be stored in GemStone.

However, you might want the classes and methods used to build the report to be stored in GemStone so that others can use them.

Certain objects can be considered your organization's *business objects*. Business objects contain the data that give your organization its strategic, competitive advantage; their instance variables contain information about the business process that they model, and their methods represent actions involved in conducting business. Keeping your business objects centralized and stored separately from the applications that access them allows your organization to serve the needs of all users, while still enforcing consistency and maintaining control of critical information.

Which objects should be secured?

What security challenges does the application pose? Determine the strategy you will use to handle those challenges. Does access to certain objects need to be restricted to only certain authorized users? Many of your business objects may fall into this category. If so, who should be authorized to access them, and how? Do your users fall into groups with different access needs? Will anyone need to execute privileged methods? The earlier you lay the groundwork for your security needs, the easier they will be to implement. Security is discussed in detail in Chapter 8.

Which objects should be connected?

Once you've decided how to partition your application objects, you will want to set up connections between the objects that will reside on the client and those that will reside on the server so that GemBuilder can automatically manage changes to them and understand how to update them properly. This connection is established by making sure a connector is defined for those objects.

A connector connects not only the immediate object but also all those objects that it references, so you don't need to define a connector for every object in your application that you want to store in the GemStone server. Instead, you should begin by identifying the subsystems in your application that define persistent objects, and then identifying a root object in each subsystem to target with a connector. You can find further discussion of connectors in Chapter 6.

How should transactions be handled?

Another decision you need to make involves transactions: when to commit and how to handle the occasional failure to commit. Do you want to use locks to ensure a successful commit? If so, identify the places in your application where you must acquire the locks. Concurrency control and locking are discussed in more detail in Chapter 7.

How can performance be improved?

If— after you have built your application— you find that its performance does not meet your expectations, you have a variety of ways to improve matters.

One of the most powerful single things you can do to improve performance is to move some of the behavior from the client to the GemStone server and let the GemStone Smalltalk execution engine do the work. This approach reduces network traffic, which is a prime cause of slow performance.

Which methods might best be executed on the GemStone server? GemStone already contains behavior for many of the common Smalltalk kernel classes and, as mentioned earlier, the syntax and class hierarchy of GemStone's Smalltalk language are so similar to those of other Smalltalks that the porting effort is likely to be relatively simple. Performance issues in general are discussed in Chapter 11. Moving execution to the GemStone server is discussed in the section entitled "Locus of Execution" on page 204.

Finally, you can configure the GemStone object server for maximum performance, given the details of your application and environment. GemStone server configuration parameters are discussed in detail in the *GemStone System Administration Guide*; in addition, the *GemStone Programming Guide* gives a variety of tips in the chapter entitled "Tuning Performance."

1.4 Delivery and Deployment

GemBuilder is provided in the form of parcels or ENVY applications named GbsRuntime, GbsTools, and CstMessengerSupport. GbsRuntime and CstMessengerSupport are required for all uses of GemBuilder. GbsTools contains development and administration tools that are normally used only during development. It is almost always desirable to have GbsTools present during development, but GbsTools can be omitted from most deployed applications.

Public and Private Classes and Methods

GemBuilder adds many classes and methods to your client Smalltalk image. Some of these we consider public, which means that they are designed to be referenced directly from your applications. GemStone avoids changing public classes and methods from release to release. Most GemBuilder classes and methods we consider private; they are used to implement the internal workings of GemBuilder and are not designed to be referenced directly from applications. Avoid using private classes and methods because they may have undocumented side effects, and because they are subject to change from release to release.

A GemBuilder class is private if its name begins with the prefix `Gbx`.

A GemBuilder method can be marked private in any of several ways:

- Any method defined in a private class is private unless the class comment indicates otherwise.
- The selectors of private methods in base class extensions begin with the prefix gbx.
- Some methods specify they are private in the method comment.
- Other methods are categorized as private in a method category marked “private”.

In general, we encourage you to use in your applications only GemBuilder classes and methods that are documented in this User's Guide. This User's Guide documents the preferred way to accomplish tasks. Other public classes or methods may be obsolete but kept for backward compatibility.

Reserved Prefix

In your code, do not define methods starting the “gb”. Methods with this prefix are reserved for GBS.

Deprecated Features

GemBuilder may include features that are deprecated in the current release. This indicates that although the feature is still functional, it may be removed in a future release. In your applications, you should eliminate any dependencies on deprecated GemBuilder features.

You can find all deprecated features of GemBuilder by browsing the senders of `#deprecated:`.

To help you determine whether your application uses any deprecated features, GemBuilder by default raises a `GemStone.Gbs.GbsInterfaceError` whenever a deprecated feature is used. The description of the `GbsInterfaceError` gives information about the deprecated feature that was being used.

If your application does not have an active handler for GbsInterfaceError when a deprecated feature is used, a walkback dialog will open. Once you've noted information about your application's use of the deprecated feature, you may proceed from the walkback or debugger, and your application will continue. This is a useful technique for a developer testing for use of deprecated features.

Once you've noted a particular use of a deprecated feature, you can avoid further walkbacks from that use until you are able to remove that use from your application. You can do this by adding a handler for the `GbsInterfaceError` that resumes the exception, at the point in your application that the deprecated feature is used.

You may also set the GemBuilder configuration parameter `deprecationWarnings` to false, as described in Appendix A. When `deprecationWarnings` is false, any code in the client image may use any deprecated GemBuilder feature without raising a `GbsInterfaceError`.

Chapter 2

Communicating with the GemStone Object Server

When you install GemBuilder, your Smalltalk image becomes "GemStone-enabled," meaning that your image is equipped with additional classes and methods that allow it to work with shared, persistent objects through a multi-user GemStone object server. Your Smalltalk image remains a single-user application, however, until you connect to the object server. To do so, your application must log in to a GemStone object server in much the same way that you log in to a user account in order to work on a networked computer system.

This chapter explains how to communicate with the GemStone object server by initiating and managing GemStone sessions.

Client Libraries
explains how to setup to use the correct client shared libraries.

GemStone Sessions
introduces sessions and explains the difference between RPC and linked sessions.

Session Control in GemBuilder
explains how to use the classes GbsSession, GbsSessionManager, and GbsSessionParameters to manage GemBuilder sessions.

The GemStone Session Browser
describes the GemStone Session Browser and Session Parameters Editor.

Logging In to and Logging Out of GemStone
describes how to log in and out of GemStone sessions programmatically and using the Session Browser.

Session Dependents
explains how to use the Smalltalk dependency mechanism to coordinate the effects of session management actions on multiple application components.

2.1 Client Libraries

Before you can log in to a GemStone object server, in addition to having GBS loaded in your image, you must have the client libraries available for loading into your image. The client libraries are provided with the GemStone object server product release. You must use the correct client libraries for the particular version of the object server you wish to connect to, and for the platform that the client Smalltalk image is running on. If you update to a new version of GBS, but continue to use the same version of the GemStone server, the same clientLibraries will be used. The client libraries must be on the platform-dependent search path.

To set the client library, use the GbsConfiguration setting “libraryName”. You may also execute:

```
GbsConfiguration current libraryName: 'libraryName'
```

See Appendix A, “GemBuilder Configuration Parameters” , for more information on this setting. To determine the correct client library name to use for your GemStone/S server product and version, see the *GemBuilder for Smalltalk Installation Guide.*

2.2 GemStone Sessions

An application connects to the GemStone object server by logging in to the server and disconnects by logging out. Each logged-in connection is known as a *session* and is supported by one Gem process. The Gem reads objects from the repository, executes GemStone Smalltalk methods, and propagates changes from the application to the repository.

Each session presents a single-user view of a multiuser GemStone repository. Most applications use a single session per client; but an application can create multiple sessions from the same client, one of which is the *current session* at any given time. You can manage GemStone sessions either through your application code or through the Session Browser.

RPC and Linked Sessions

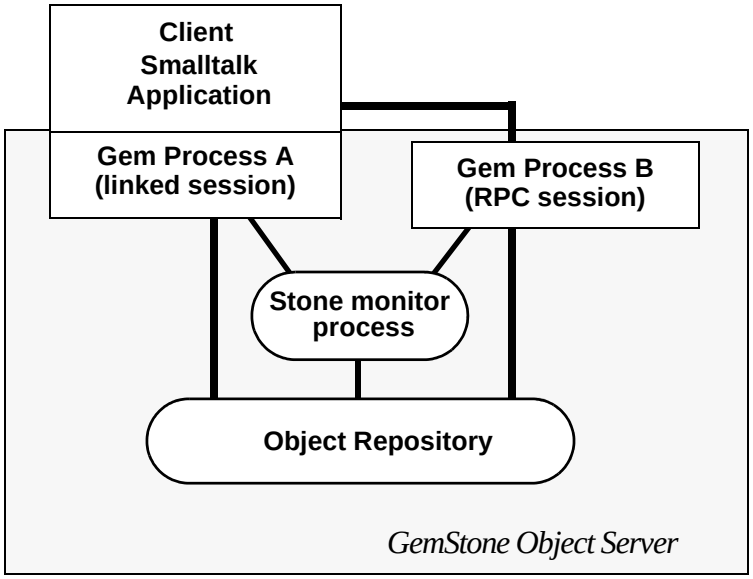
A Gem can run as a separate operating system process and respond to Remote Procedure Calls (RPCs) from its client, in which case the session it supports is called an *RPC* session.

On platforms that host the GemStone object server and its runtime libraries, one Gem can be integrated with the application into a single operating system process. That Gem is called a *linked* session. When running linked, an application and its Gem must run on the same machine and the runtime code requires additional memory.

An RPC session offers more flexibility because the application and its Gem are separate processes that can run on different hosts in a network. Any GemBuilder client can create RPC sessions. Where a linked session is supported, an application can create multiple sessions, but only one can be linked. To suppress linked sessions, forcing all Gems to run as RPC processes, you can load the RPC-only version of the shared libraries.

Figure 2.1 shows an application with two logged-in sessions. Gem A is a linked session, sharing the client Smalltalk application's process space, while Gem B is an RPC session running as a separate process.

Figure 2.1 RPC and Linked Gem Processes



2.3 Session Control in GemBuilder

Managing GemStone sessions involves many of the same activities required to manage user sessions on a multi-user computer network. To manage GemStone sessions, you need to do various operations:

- Identify the object server to which you wish to connect.
- Identify the user account to which you wish to connect.
- Log in and log out.
- List active sessions.
- Designate a current session.
- Send messages to specific sessions.

Three GemBuilder classes provide these session control capabilities: GbsSession, GbsSessionParameters, and GbsSessionManager.

GbsSession

An instance of GbsSession represents a GemStone session connection. A successful login returns a new instance of GbsSession. You can send messages to an active GbsSession to execute GemStone code, control GemStone transactions, compile GemStone methods, and access named server objects.

GbsSessionParameters

Instances of GbsSessionParameters store information needed to log in to GemStone. This information includes the Stone name, your user name, passwords, and the set of connectors to be connected at login.

GbsSessionManager

There is a single instance of GbsSessionManager, named GBSM. Its job is to manage all GbsSessions logged in from this client, support the notion of a current session (explained in the following section), and handle other miscellaneous GemBuilder matters. Whenever a new GbsSession is created, it is registered with GBSM. GBSM shuts down any server connections before the client Smalltalk quits.

Session Parameters

To initiate a GemStone session, you must first identify the object server and user account to which you wish to connect. This information is stored in an instance of GbsSessionParameters and added to a list maintained by GBSM. You can provide the information through window-based tools or programmatically. Both methods are described in later sections. In either case, you must supply these items:

- **The name of the GemStone repository**
  For a Stone running on a remote server, be sure to include the server's hostname in Network Resource String (NRS) format. For instance, for a Stone named "gs64stone" on a node named "pelican", specify an NRS string of the form:

  ```
  !@pelican!gs64stone
  ```

  (Appendix C describes NRS syntax in detail.)

- **GemStone user name and GemStone password**
  This user name and password combination must already have been defined in GemStone by your GemStone data curator or system administrator. (GemBuilder provides a set of tools for managing user accounts—see "User Account Management Tools" on page 177.) Because GemStone comes equipped with a data curator account, we show the DataCurator user name in many of our examples.

- **Host username and Host password** (not required for a linked session, or if netldi is run in guest mode)
  This user name and password combination specifies a valid login on the Gem's host machine (the network node specified in the Gem service name, described below). Do not confuse these values with your GemStone username and password. You do not need to supply a host user name and host password if you are starting a linked Gem process. However, an application that must control more than one GemStone session can use a linked interface for only one session. Other sessions must use the RPC interface.
- **Gem service** (not required for a linked session)
  The name of the Gem service on the host computer (that is, the Gem process to which your GemBuilder session will be connected). For most installations, the Gem service name is `gemnetobject`.

  You can specify that the gem is to run on a remote node by using an NRS for the Gem service name For example:

  ```
  !@pelican!gemnetobject
  ```

  You do not need to supply a Gem Service name if you are starting a linked Gem process.

Once defined, an instance of GbsSessionParameters can be used for more than one session. Thus, a session description that includes the RPC-required parameters can be used for both linked and RPC logins.

Defining Session Parameters Programmatically

The instance creation method for a full set of RPC parameters is:

```
GbsSessionParameters newWithGemStoneName: aGemStoneName
  username: aUsername
  password: aPassword
  hostUsername: aHostUsername
  hostPassword: aHostPassword
  gemService: aGemServiceName
```

For a shorter set of parameters that supports only linked logins, you can use a shorter creation method:

```
GbsSessionParameters newWithGemStoneName: aGemStoneName
  username: aUsername
  password: aPassword
```

Storing Session Parameters for Later Use

If you want the GemBuilder session manager to retain a copy of your newly-created session description for future use, you must register it with GBSM:

```
GBSM addParameters: aGbsSessionParameters
```

Once registered with GBSM and saved with the image, the parameters are available for use in future invocations of the image. In addition, the Session Browser and other login prompters make use of GBSM's list of session parameters.

Executing the expression `GBSM knownParameters` returns an array of all GbsSessionParameters instances registered with GBSM.

To delete a registered session parameters object, send `removeParameters:` to GBSM:

```
GBSM removeParameters: aGbsSessionParameters
```

Password Security

You can control the degree of security that GemBuilder applies to the passwords in a session parameters object. For example, when you create the parameters object, you can specify the passwords as empty strings. When the parameters object is subsequently used in a login message, GemBuilder will prompt the user for the passwords.

For example:

```
mySessionParameters := GbsSessionParameters
   newWithGemStoneName: '!@pelican!gs64stone'
   username:            'DataCurator'
   password:            ''
   hostUsername:        'lisam'
   hostPassword:        ''
   gemService:          '!@pelican!gemnetobject'
```

If convenience is more important than security, you can fill in the passwords and then instruct the parameters object to retain the password information for future use:

```
mySessionParameters rememberPassword: true;
                    rememberHostPassword: true
```

The default "remember" setting for both passwords is `false`, which causes the stored passwords to be erased after login.

2.4 The GemStone Session Browser

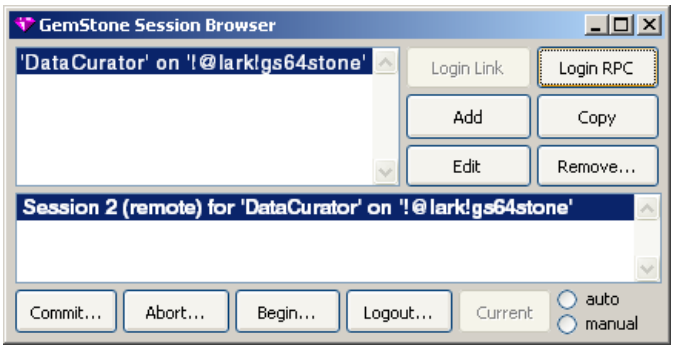
The GemStone Session Browser streamlines logging in and logging out of GemStone and managing sessions and transactions. This section explains how to invoke the Session Browser, and how to use it to define session parameters and to log in and out of GemStone.

Starting the Session Browser

1. Start your GemBuilder for Smalltalk image.
2. Select **Browse Sessions** from the GemStone menu to open a Session Browser.

   Figure 2.2 shows the Session Browser.

Figure 2.2 The GemStone Session Browser

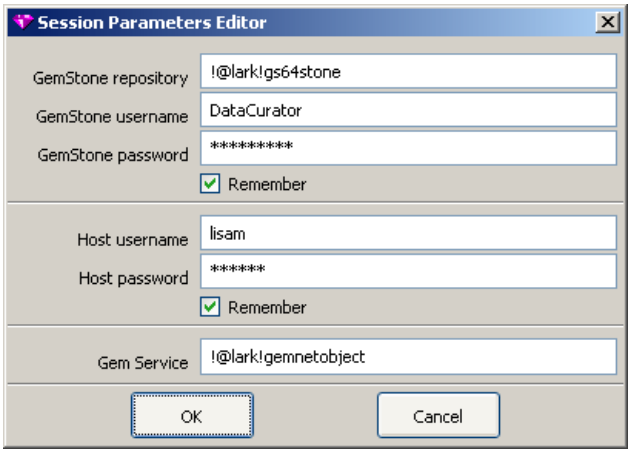


Opening the Session Parameters Editor

Select the **Add** button to define a set of session parameters. A Session Parameters Editor appears, as shown in Figure 2.3.

The first time this is done in a new image, the server-specific client libraries are loaded. Any problems in the client library configuration will show up now.

Figure 2.3 The Session Parameters Editor



Use the **Tab** key or the mouse to move through the fields in the login dialog, and the **Return** key to accept input or changes in the login dialog. Provide the session parameters described previously (see "Session Parameters" on page 35). For maximum password security, leave the **Password** and **Host Password** fields empty, and the **Remember** boxes unselected.

When you click on **OK**, GemBuilder creates an instance of GbsSessionParameters and registers it with GBSM. The new session description is added to the Session Browser.

To change a session parameters object, select the name of the parameters object in the upper left pane of the Session Browser and use the browser's **Edit** button to open a Session Parameters Editor. Use the Session Parameters Editor to change existing session parameters; clicking on **OK** causes your changes to take effect.

Managing Session Parameters

Using the Session Browser buttons or the pop-up menu accessible with the *operate* mouse button, you can manage your set of session parameters.

The Session Browser supports the following operations:.

Table 2.1 Functions in the Session Browser

Add	Open an empty Session Parameters Editor.
Copy	Make a copy of the selected session parameters, and add it to the list.
Edit	Open a Session Parameters Editor on the selected session parameters.
Login link	Log in linked, using the selected session parameters.
Login Rpc	Log in RPC, using the selected session parameters.
Remove...	Remove the selected session parameters
Import...	Prompt for a filename, and read a previously exported list of Session Parameters from a text file. (only on pop-up menu)
Export...	Prompt for a filename, and write out the list of Session Parameters to a text file. **The exported information is in plain text. If you have included passwords in your session parameters, these will be visible.** (only on pop-up menu)

In addition, if you have many session parameters, you can drag and drop a selected session parameters to order them within the list. This has no affect on how they are used, but can make frequently used ones easier to see.

2.5 Logging In to and Logging Out of GemStone

Before you can start a GemStone session, you need to have a Stone process and, for an RPC session, a NetLDI (network long distance information) process running.

Depending on the terms of your GemStone license, you can have many sessions logged in at once from the same GemBuilder client. These sessions can all be attached to the same GemStone repository, or they can be attached to different repositories.

Programmatic Session Management

Logging In to GemStone

The protocol for logging in is understood both by GBSM and by instances of GbsSessionParameters. To log in using a specific session parameters object, send a `login` message to the parameters object itself:

```
mySession := aGbsSessionParameters login
```

To start multiple sessions with the same parameters, simply repeat these login messages.

An application can also send a generic login message to GBSM:

```
mySession := GBSM login
```

This message invokes an interactive utility that allows you to select among known GbsSessionParameters or to create a new session parameters object using the Session Parameters Editor.

A successful login returns a unique instance of GbsSession. (An unsuccessful login attempt returns `nil`.) Each instance of GbsSession maintains a reference to that session's parameters, which you can retrieve by sending:

```
myGbsSessionParameters := aGbsSession parameters
```

GBSM maintains a collection of currently logged in GbsSessions. You can determine if any sessions are logged in with `GBSM isLoggedIn` and you can execute `GBSM loggedInSessions` to return an array of currently logged in GbsSessions.

The Current Session

When a new GbsSession is created, it is registered with GBSM, which maintains a variable that represents the *current* session. When a session logs in, it becomes the current session. If you execute code in a GemStone tool, the code is evaluated in the session that was current when you opened that tool. If you send a message to GBSM that is intended for a session, the message is forwarded to the current session.

You can send a message directly to any logged-in GbsSession, even when it is not the current session. If you send a specific session a message that executes code, that code is evaluated in the receiving session, regardless of whether it is the current session.

Most applications have only one session logged in at a time. In this case, that session will always be the current session, and it is safe to send messages to GBSM for forwarding to the session.

However, if your application concurrently logs in more than one session, your application should send messages directly to each session. If your application client uses multiple Smalltalk processes it is very difficult to accurately coordinate the changing of the current session.

Sending the message `GBSM currentSession` returns the current GbsSession. You can change the current session in a workspace by executing an expression of the following form:

```
GBSM currentSession: aGbsSession.
```

Your application can make another session the current session by executing code like that shown in Example 2.1:

Example 2.1

```
|s1 s2|
 s1 := GBSM login.
 s2 := GBSM login.
GBSM currentSession: s1.      "Make s1 current"
 .
 .                            "Do some work"
 .
GBSM currentSession: s2.      "Make s2 current"
```

Each GemStone browser, inspector, debugger, and breakpoint browser is attached to the instance of GbsSession that was the current session when it opened. For example, you can have two browsers open in two different sessions, such that operations performed in each browser are applied only to the session to which that browser is attached.

Workspaces, however, are not session-specific. GS-Do its executed in a workspace default to the current session.

Logging Out of GemStone

To instruct a session to log itself out, send logout to the session object:

```
aGbsSession logout
```

Or, you can execute the more generic instruction:

```
GBSM logout
```

This message prompts you with a list of currently logged-in sessions from which to choose.

Before logging out, GemBuilder prompts you to commit your changes, if the GbsConfiguration setting confirm is true (it is true by default). If you log out after performing work and do not commit it to the permanent repository, the uncommitted work you have done will be lost.

If you have been working in several sessions, be sure to commit only those sessions whose changes you wish to save.

Session Management Using the Session Browser

You can use the Session Browser to perform the same session management tasks that you can perform programmatically: log in to the GemStone server, view current sessions, set the current session, and log out of the GemStone server.

Logging In to GemStone

To log into the GemStone server with the Session Browser, select the name of the session parameters object in the upper left pane, and click on either **Login Lnk** or **Login Rpc.**

When you are logged in, the Session Browser displays the session description in its lower pane.

If your login is not successful, make sure you entered the correct parameters and that the necessary server processes are running.

Setting the Current Session

The Session Browser's upper pane shows all of the known parameters that are registered with GBSM. The lower pane shows all sessions currently logged in.

To change the current session, select a logged-in session in the lower pane and click the **Current** button.

Logging Out of GemStone

To log out of GemStone from the Session Browser, select the session in the browser's lower pane and click on **Logout** in the row of buttons at the bottom of the browser.

Before logging out, GemBuilder prompts you to commit your changes, if the GbsConfiguration setting confirm is true (it is true by default). If you log out after performing work and do not commit it to the permanent repository, the uncommitted work you have done will be lost.

If you have been working in several sessions, be sure to commit only those sessions whose changes you wish to save.

2.6 Session Dependents

An application can create several related components during a single GemBuilder session. When one of the components commits, aborts, or logs out, the other components can be affected and so may need to coordinate their responses with each other. In the GemBuilder development environment, for example, you can commit by clicking on a button in the Session Browser. But before the commit takes place, all other session-dependent components are notified that a commit is about to occur. So a related application component, such as an open browser containing modified text, prompts you for permission to discard its changes before allowing the commit to proceed.

Through the Smalltalk dependency mechanism, any object can be registered as a dependent of a session. In practice, a session dependent is often a user-visible application component, such as a browser or a workspace. When one application component asks to abort, commit, or log out, the session asks all of its registered dependents to approve before it performs the operation. If any registered dependent vetos the operation, the operation is not performed and the method (`commitTransaction`, `abortTransaction`, etc.) returns `nil`.

To make an object a dependent of a GbsSession, send:

mySession `addDependent:` *myObj*

To remove an object from the list of dependents, send the following message:

mySession `removeDependent:` *myObj*

So, for example, a browser object might include code similar to Example 2.2 in its initialization method:

Example 2.2

```
| mySession |
mySession := self session.
"Add this browser to the sessions dependents list"
(session dependents includes: self)
      ifFalse: [session addDependent: self]
...
```

When a session receives a commit, abort, or logout request, it sends an `updateRequest:` message to each of its dependents, with an argument describing the nature of the request. Each registered object should be prepared to receive the `updateRequest:` message with any one of the following aspect symbols as its argument:

#queryCommit
The session with which this object is registered has received a request to commit. Return `true` to allow the commit to take place or `false` to prevent it.

#queryAbort
The session with which this object is registered has received a request to abort. Return `true` to allow the abort to take place or `false` to prevent it.

#queryEndSession
The session with which this object is registered has received a request to terminate the session. Return `true` to allow the logout to take place or false to prevent it.

Example 2.3 shows how a session dependent might implement an `updateRequest:` method.

Example 2.3

```
updateRequest: aspect

"The session I am attached to wants to do something.
 Return a boolean granting or denying the request."

^(#(queryAbort queryCommit queryEndSession)
  includes: aspect)
    ifTrue: [
             "My session wants to commit or abort.
              OK unless user doesn't want to."
   self askUserForPermission ]
   ifFalse: [
             "Let any other action occur."
    true]
```

After the action is performed, the session sends `self changed:` with a parameter indicating the type of action performed. This causes the session to send an `update:` message to each of the registered dependents with one of the following aspect symbols:

#committed
All registered objects have approved the request to commit, and the transaction has been successfully committed.

#aborted
All registered objects have approved the request to abort, and the transaction has been aborted.

#sessionTerminated
The request to log out has been approved and the session has logged out.

Each registered dependent should be prepared to receive an `update:` message with one of the above aspect symbols as its argument. Example 2.4 shows how a session dependent might implement an `update:` method.

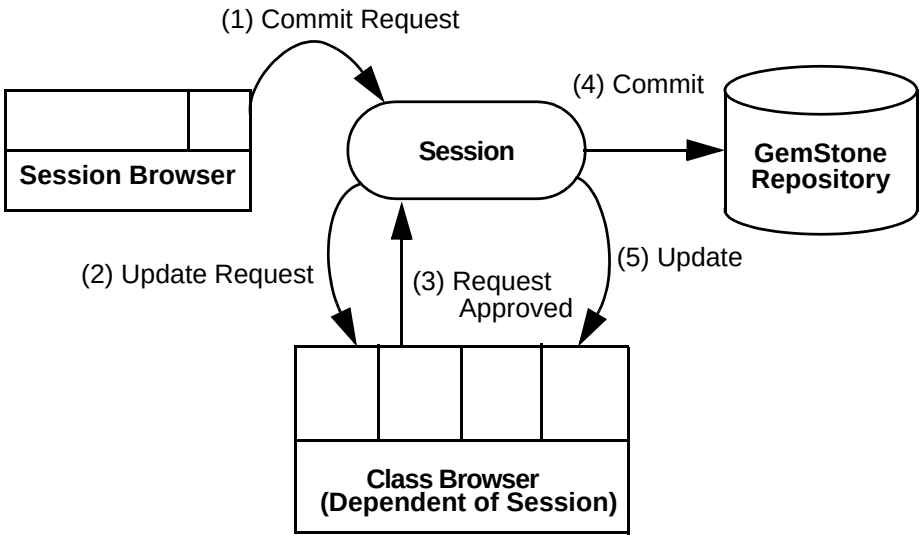
Example 2.4

```
update: aSymbol
"The session I am attached to just did something.
 I might need to respond."

(aSymbol = #sessionTerminated) ifTrue: [
"The session this tool is attached to has logged out
 - close ourself."
self builder notNil ifTrue:
      [self closeWindow]]
```

Figure 2.4 summarizes the sequence of events that occurs when a session queries a dependent before committing. In the figure, the Session Browser sends a commit request (`commitTransaction`) to a session (1). The session sends `updateRequest: #queryCommit` to each of its dependents (2). If every dependent approves (returns **true**), the commit proceeds (4). Following a successful commit, the session notifies its dependents that the action has occurred by sending `update: #committed` to each (5).

Figure 2.4 Committing with Approval From a Session Dependent



Chapter 3

Using the GemStone Programming Tools

After you install GemBuilder, many menus in your Smalltalk image contain additional commands for executing GemStone Smalltalk code and accessing GemBuilder programming tools. These tools are in many ways similar to those of the client Smalltalk, but with important differences; this chapter describes those differences.

GemStone Menu
introduces the tools and options available from the GemStone menu.

Browsing Code
describes the GemStone Classes Browser and other code browsers.

System Workspace
describes the System Workspace.

Coding
explains how to use the GemBuilder tools to create classes and methods in GemStone Smalltalk for execution and storage on the server.

3.1 GemStone Menu

The **GemStone** menu accessible from various tools, gives you access to the GemStone Smalltalk compiler and the GemBuilder programming tools. Many of these functions are also available from pop-up menus in the browsers and tools.

As shown in Table 3.1, the **GemStone** menu provides commands for executing GemStone Smalltalk code and accessing the GemStone programming tools.

Table 3.1 The GemStone Menu

Browse Sessions	Opens a GemStone Session Browser, allowing you to log into or out of the GemStone server and manage transactions. The Session Browser is described in Chapter 2.
Browse Connectors	Opens a GemStone Connector Browser, allowing you to manage the connections between GemStone server and Smalltalk client objects. The Connector Browser is described in Chapter 6.
Browse	Produces a submenu with the following options:
All Classes	Opens a GemStone Browser, comparable to the client Smalltalk System or Classes Browser. The GemStone Browser is described in "Browsing Code" on page 52.
Class...	Prompts for the name of a class, then opens a browser focused on that class.
Namespace...	Prompts for the name of a symbol dictionary, then opens a browser focused on that dictionary.
Senders of...	Prompts for the name of a message selector, then opens a method browser showing senders of that message.
Implementors of...	Prompts for the name of a message selector, then opens a method browser showing implementors of that message.
References to...	Prompts for the name of a variable, then opens a method browser showing all methods that refer to that variable.

Table 3.1 The GemStone Menu(Continued)

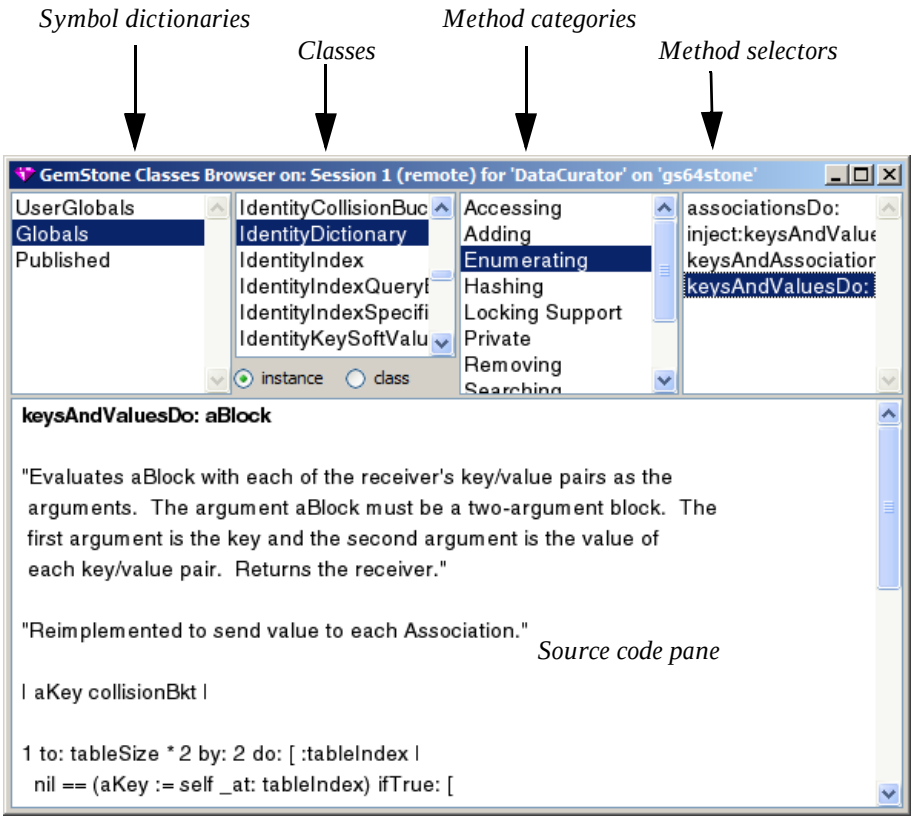
	Methods with substring...	Prompts for a string, then opens a method browser showing all methods whose source contains that string.
Admin	Produces a submenu with the following options:	
	Users	Opens the GemStone User Account Management Tools, allowing you to create new users, assign attributes to them, and manage user accounts, provided you have the privileges to do so. The User Account Management Tools are described in Chapter 8.
	Symbol Lists	Opens a Symbol List Browser, allowing you to examine and modify symbol dictionaries and their entries. The Symbol List Browser is described in Chapter 8.
	Segments	Opens a Segment Tool, allowing you to control authorization at the object level by assigning objects to segments. The Segment Tool is described in Chapter 8.
Tools	Produces a submenu with the following options:	
	Settings	Opens a Settings Browser in which you can examine, change, and store parameters for configuring GemBuilder. The Settings Browser is described in Chapter 11.
	Breakpoints	Opens a Breakpoint Browser, allowing you to set and clear breakpoints in GemStone Smalltalk code. The Breakpoint Browser is described on page 76.
	System Workspace	Opens the GemStone System Workspace, a workspace containing a variety of useful GemStone Smalltalk and client Smalltalk expressions.
About GemBuilder	Opens a window providing the GemBuilder version and copyright information.	

3.2 Browsing Code

After logging in to GemStone, open a GemStone Classes Browser by choosing **GemStone > Browse > All Classes**.

The GemStone Classes Browser allows you access source and other information about each of the kernel classes and methods; you can also create GemStone Smalltalk classes and methods in the GemStone repository.

Figure 3.1 GemStone Classes Browser



The GemStone Classes Browser is similar to the client Smalltalk System or Classes Browser, but a few differences exist: for example, the upper left pane contains a list of symbol dictionaries, GemStone's mechanism for implementing

namespaces. This facilitates finding and sharing objects efficiently. The symbol dictionaries that you can access are listed in the GemStone Browser's symbol list pane.

When you select a symbol dictionary in the Symbol List pane, all classes defined in that dictionary appear in the Classes pane to the right. (Symbols other than classes can be viewed by opening an inspector on the symbol dictionary in question, or by selecting **`GemStone > Admin > Symbol Lists`**.)

GemStone Smalltalk categorizes methods by function to make them easier to browse. When you select a class in the Classes pane, a list of its method categories appears in the Method Categories pane to the right.

When you select a method category, all the message selectors in that category appear in the rightmost Method Selectors pane.

As in the comparable client Smalltalk browsers, you can switch focus between **instance** or **class** methods using the toggle or radio buttons provided.

Also as in the comparable client Smalltalk browsers, when you select a method, its source code is displayed in the lower portion of the browser—the source pane. In this pane, you can edit and recompile the method, set breakpoints in it, or execute fragments of GemStone Smalltalk code as in a workspace.

Each pane of the GemStone Browser has pop-up menus accessible with the *operate* mouse button. The GemStone-specific commands in the popup menus of the Symbol List, Class, Categories, and Methods panes are described below.

Symbol List Pane

The GemStone-specific menu options available in the Symbol List pane shown in Table 3.2.

Table 3.2 Symbol List Menu in GemStone Browser

file out as...	Prompts you for a file name under which to save all the class and method definitions for all the classes in the selected SymbolDictionary.
file out methods as...	Prompts you for a file name under which to save all the methods in all the classes in the selected SymbolDictionary, so that you can file them into a client class without creating a new version of the class.
spawn	Open a Dictionary Browser on the selected SymbolDictionary.
inspect	Open an inspector on the selected SymbolDictionary.
add...	Add a new Symbol Dictionary
rename as...	Rename the selected SymbolDictionary.
remove...	Remove the selected SymbolDictionary. Do not remove Globals.
update	Update the view of the browser to the current state in the image.
commit	Attempts to commit modifications to the repository that occurred during the current GemStone transaction.
abort	Undoes all changes that you have made in the repository since the beginning of the current transaction. You are asked to confirm this choice.
find class...	Navigate to a specific class by name. The search string is case sensitive, and can include wild cards.

Class Pane

The GemStone-specific menu commands available in the GemStone Browser's Class Pane are described in Table 3.3. A later section discusses the procedure required to add the definition of a new GemStone class to the currently selected symbol dictionary.

Table 3.3 Class Menu in GemStone Browser

file out as...	
file out methods as...	Prompts you for a file name under which to save all the methods of the selected class, so that you can file them into a client class without changing the client class's structure.
browse class	Open a class browser on the selected class.
browse hierarchy	Open a hierarchy browser on the selected class.
browse versions	Open a class version browser on the selected class.
browse references	Open a method list on all references to the current version of the class, or all versions of the class.
hierarchy	Display the class hierarchy in the text pane.
definition	Display the class definition in the text pane (the default)
comment	Display the class comment in the text pane.
move to...	Prompt for another SymbolDictionary to which to move the selected class.

Table 3.3 Class Menu in GemStone Browser(Continued)

remove...	Remove the selected class.
create access	Creates methods for accessing and updating the instance variables of the selected class.
create in ST	Creates a client Smalltalk class having the same name and structure as the selected GemStone Smalltalk class, if one doesn't already exist. If it does exist, executing this menu item has no effect.
compile in ST	Creates a client Smalltalk class having the same name and structure as the selected GemStone Smalltalk class, and compiles all currently defined methods for the class. If necessary, a notifier lists any methods that cannot be compiled in client Smalltalk.

Pop-up Text Pane Menu

A pop-up menu appears in any text pane when you press the *operate* mouse button. This menu provides the same commands as the corresponding menu in the client Smalltalk browser's text pane. In addition, it contains menus for displaying, executing, inspecting, and filing in GemStone Smalltalk code and for using breakpoints in GemStone Smalltalk code.

The GemStone-specific commands available from a text area pane are shown in Table 3.4.

Table 3.4 Pop-up Menu in GemStone Browser's Text Pane

GS-Do it	Executes the code in GemStone.
GS-Print it	Executes the code in GemStone and displays the result in the text area.
GS-Inspect it	Executes the code in GemStone and opens an inspector on the result.
set break	Sets a breakpoint at the step point nearest the cursor location. If the cursor is not exactly at a step point, scans the method from the current cursor location on and sets a breakpoint at the next step point. See "Debugging" on page 69 for a full discussion of using breakpoints.

GemBuilder also adds the following items to the appropriate menus in the client Smalltalk browsers:

Table 3.5 Additional GemStone Menu Items

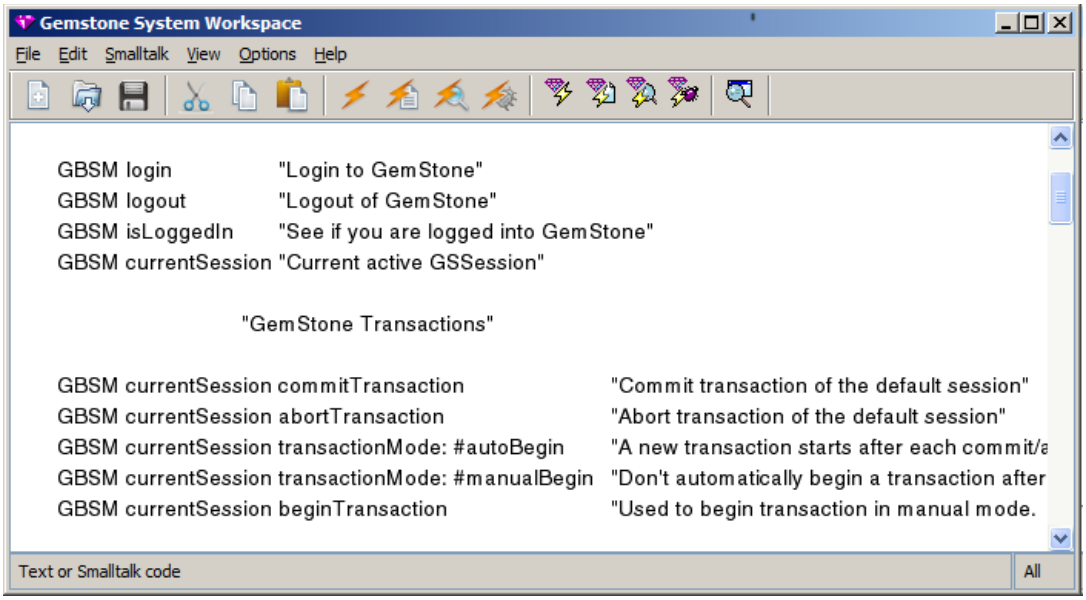
create in GS	Creates a GemStone Smalltalk class having the same name and structure as the selected client Smalltalk class, if one doesn't already exist. If it does exist but you've changed its structure, executing this menu item creates a new version of the class.
compile in GS	Creates a GemStone Smalltalk class having the same name and structure as the selected client Smalltalk class, and compiles all currently defined methods for the class in GemStone. If necessary, a notifier lists any methods that cannot be compiled.

3.3 The System Workspace

The GemStone System Workspace is a workspace containing templates for many useful GemStone Smalltalk and client Smalltalk expressions. Browse it to familiarize yourself with its contents.

To open a GemStone System Workspace (Figure 3.2), choose **GemStone > Tools > System Workspace** from the **GemStone** menu.

Figure 3.2 GemStone System Workspace


Gemstone System Workspace
File Edit Smalltalk View Options Help
GBSM login "Login to GemStone"
GBSM logout "Logout of GemStone"
GBSM isLoggedIn "See if you are logged into GemStone"
GBSM currentSession "Current active GSSession"
"GemStone Transactions"
GBSM currentSession commitTransaction "Commit transaction of the default session"
GBSM currentSession abortTransaction "Abort transaction of the default session"
GBSM currentSession transactionMode: #autoBegin "A new transaction starts after each commit/a
GBSM currentSession transactionMode: #manualBegin "Don't automatically begin a transaction after
GBSM currentSession beginTransaction "Used to begin transaction in manual mode.
Text or Smalltalk code
All

3.4 Coding

This section explains how to define new GemStone classes and methods, and describes aspects of coding unique to GemStone Smalltalk.

About GemStone Smalltalk Classes

The following discussion summarizes the main differences between GemStone Smalltalk and client Smalltalks. For complete information about programming in GemStone Smalltalk, refer to the *GemStone Programming Guide.*

Constraints

Constraints are supported in GemStone / S, but are not available in GemStone / S 64 Bit.

Constraints constrain the value of an instance variable to contain only specified kinds of objects. Constraining a variable means that its value is always either an instance of the specified class, a subclass thereof, or `nil`.

Constraints can be circular: you can constrain an instance variable to be an instance of its own class, or you can also constrain instance variables of two classes to each hold instances of the other.

Constraints are inherited: when you define a subclass, its inherited instance variables by default bear the same constraints as those specified in their superclass. However, inherited instance variables can be further constrained in a subclass. In this case, the instance variable's new constraint must be a subclass of that specified by the inherited constraint.

To further constrain inherited instance variables, specify the name of the inherited variable and its new constraint in the argument to the `constraints`: keyword in the class definition template. For example, suppose you have defined a class Employee with instance variables named `jobTitle` and `department` that are constrained to be Strings. You can now create a subclass of Employee named FormerEmployee and constrain the inherited variables `jobTitle` and `department` to be InvariantStrings. FormerEmployee's new instance variables can be constrained or not, as you require, and its other inherited instance variables retain whatever constraints were set in the superclass that defined them, if any.

Invariance

Instances can be invariant. A class definition can specify that all instances are invariant, meaning they can be modified only during the transaction in which it is created. After the transaction is committed, you can no longer modify its instance variables, nor the size or class of the object.

Specify invariance for a class by providing the argument *true* to the `instancesInvariant:` keyword in the class definition template.

Class-level invariance is useful for supporting literals in methods and in other limited situations, but it is generally more cumbersome than object-level invariance. Any object can be made invariant by sending it the message `immediateInvariant`. This mechanism protects objects from being modified and can be useful for maintaining the integrity of your repository. After `immediateInvariant` is sent to an object, you can no longer modify its instance variables, nor the size or class of the object. The effect of the `immediateInvariant` message is not reversible.

The message `isInvariant` returns true if the receiver is invariant; false otherwise.

Defining a New Class

To define a new GemStone class:

Step 1. Open a GemStone Browser if one is not already open.

Step 2. In the Symbol List pane, select the dictionary in which you wish to refer to the new class. Make sure no class is selected in the class list.

The browser displays the class definition template:

```
NameOfSuperclass subclass: 'NameOfClass'
   instVarNames: #('instVarName1' 'instVarName2')
   classVars: #('ClassVarName1' 'ClassVarName2')
   classInstVars #('ClassInstVarName1' 'ClassInstVarName2')
   poolDictionaries: #[]
   inDictionary: aDictionary
   constraints: #[]
   instancesInvariant: false
   isModifiable: false
```

This is the basic form of the subclass creation message in GemStone Smalltalk.

Step 3. Replace *NameOfSuperclass* with the name of your new class's immediate superclass.

Step 4. Replace *NameOfClass* with the name of the new class. By convention, the first letter of each GemStone class name is capitalized.

Step 5. Replace *instVarName* with the names of any instance variables, or delete all the text within the parentheses if your new class has no instance variables.

A class can define up to 255 named instance variables.

Step 6. Replace *classVarNames* with the names of any class variables, or delete all the text within the parentheses if your new class has no class variables.

Step 7. Replace *classInstVarNames* with the names of any class instance variables, or delete all the text within the parentheses if your new class has no class instance variables.

Step 8. Fill in the brackets after the `poolDictionaries:` keyword with any pool dictionaries that you want the class to access. Pool dictionaries are special-purpose storage structures that enable any arbitrary group of classes and their instances to share information. When classes share a pool dictionary,

methods defined in those classes can refer to the variables defined in the pool dictionary.

Step 9. After the `inDictionary:` keyword, the name of the selected symbol dictionary is inserted in the template. This is the symbol dictionary that will allow you to refer to your class by name. Unless you replace the inserted text with the name of another symbol dictionary to which you have access, your new class is defined in the selected symbol dictionary.

Step 10. With GemStone/S 64 Bit, any arguments to this keyword are not used.

With GemStone/S, fill in the brackets after the `constraints:` keyword with any constraints you wish to specify for one or more of the instance variables.

To constrain the elements of a collection, type the name of the constraint class inside the brackets. For example, to constrain the Bag subclass BagOfEmployees to contain only instances of the class Employee, type:

```
constraints: #[Employee]
```

To constrain a named instance variable, type the name of the variable and the constraint class as a pair separated by a comma, each within its own set of square brackets, also separated by commas. Preface the instance variable name with a #. For example, to constrain the instance variables `name` and `address` of the class Employee to be a String and an instance of the class Address, respectively, type:

```
constraints: #[ #[ #name, String], #[ #address, Address] ]
```

Step 11. After the `instancesInvariant:` keyword, specify whether instances of the class are modifiable. The default is `false`—change this to `true` if you wish instances to be invariant.

Step 12. After the `isModifiable:` keyword, specify whether the structure of the class can be modified. The default value is `false`—change this to `true` if you wish class to be invariant.

Step 13. Accept or save your changes and commit your transaction to make the class part of the repository.

NOTE

You cannot subclass certain GemStone kernel classes. To determine which ones, execute the method `subclassesDisallowed` *against the class Object. The method returns* `true` *for any class that you cannot subclass.*

For example, consider the following definition of a class named Employee:

Example 3.1

```
Object subclass: 'Employee'
      instVarNames: #( 'name' 'employeeNum' 'jobTitle' 'department'
                       'address')
      classVars: #()
      classInstVars #()
      poolDictionaries: #[]
      inDictionary: UserGlobals
```

Employee resides in the developer's UserGlobals dictionary.

```
      constraints: #[]
      instancesInvariant: false
```

When instances of the class are created, their values will be modifiable even after they've been committed to the repository.

```
      isModifiable: true
```

This class is modifiable; instance variables can still be added, removed, and constrained, and class or class instance variables can be added. However, as long as the class itself remains modifiable, no instances of it can be created.

Subclass Creation Methods

You can choose from a variety of subclass creation messages, depending on the type of class you want to create. For example, to create a byte subclass, replace the initial keyword `subclass:` with the keyword `byteSubclass:`. If the superclass is not a subclass of String, instances of the new class store and return SmallIntegers in the range 0–255.

Similarly, if you wish to create an indexable subclass, replace the initial keyword `subclass:` with the keyword `indexableSubclass:`. Instances of the new class are represented as pointer objects.

For complete descriptions of the different kinds of classes, see the discussion of class storage formats in the chapter entitled "Advanced Class Protocol" in the *GemStone Programming Guide.*

If you wish to set the class history of your new class explicitly, you can include the keyword `newVersionOf:` in the class definition template or any subclass creation message, after `instancesInvariant:` and before `isModifiable:`. If the argument to this keyword is a class, this method creates the new class as a new version of that class, and the two classes share a class history. In this way, you can make one class a new version of another even if they do not have the same name.

If the argument to the `newVersionOf:` keyword is `nil`, the new class is created with a new class history.

If you do not include the `newVersionOf:` keyword, the compiler checks to see if another class having the same name already exists. If it does, the new class is compiled as a new version of the other class and shares its class history. If it does not, the new class is created with a new class history.

For more discussion of class versions and histories, see the chapter entitled "Class Versions and Instance Migration" in the *GemStone Programming Guide.*

Private Instance Variables

Some GemStone kernel classes have private instance variables. For example, the superclass of GemStone Bag class defines four, used by the object manager and primitives to implement features of nonsequenceable collections, such as adding indexing structures for efficient querying. Private instance variable names begin with an underscore (_). When defining subclasses, private instance variables cannot be modified or constrained.

Modifying an Existing Class

If you select an existing GemStone Smalltalk class, then modify and save the class definition, you create a new version of that class and all of its subclasses. The browser attempts to recompile all methods from the previous version into the new version. Methods that fail to recompile are presented in a method list browser, from which you can correct the errors. If the class has subclasses, they are also versioned and their methods recompiled.

Versioning a class does not migrate its instances; they're still instances of the old class. You can migrate some or all instances of one version of a class to another version explicitly.

For more information on migrating instances, see the chapter entitled "Class Versions and Instance Migration" in the *GemStone Programming Guide.*

NOTE
You can only modify classes for which you have write authorization

To create a new version of a class:

Step 1. Select the class in the browser to bring up its definition in the source pane.

Step 2. Edit the definition as required.

Step 3. Select **Save** or **accept** from the pop-up menu.

Whenever you create a class with the same name as a class that already exists in one of your symbol dictionaries, the new class is automatically created as the latest version of the existing class and it automatically shares the same class history. Instances created after the redefinition have the new class's structure and access the new class's methods. Instances that were created earlier have the old class's structure and access the old class's methods, but they can be migrated to the new class.

Let's assume that you have a class named Employee with instance variables for `name`, `employeeNum`, `jobTitle`, `department`, and `address`, and that the class is defined as shown in Figure 3.1. Suppose that you decide that the class needs an additional instance variable named `salary` to represent the Employee's salary.

To do this, you can define a new version of the class Employee to include the new instance variable. Keeping the same name as the old class ensures that it shares the same class history as the previous version.

After you compile the class definition, the new class is named Employee, and all of the original instance and class methods are copied to the new class. Any existing instances will still belong to the original class and may have to be migrated to the new class. (See "Instance Migration Within GemStone" on page 190.)

Defining Methods

You can modify only methods for which you have write authorization— for example, methods that you have written for your own classes. You cannot modify any GemStone kernel class method—that is, any method that is defined for one of the predefined classes supplied with the GemStone system.

Public and Private Methods

GemStone has both public and private methods. *Public* GemStone methods are supported. *Private* GemStone methods are those implemented to support the public protocol—they are not supported and are subject to change.

Private GemStone methods are those whose selector is prefixed with an underscore (_). They appear in the browsers along with the public methods, and you can display the source for them.

CAUTION

Private methods are subject to change at any time. Do not depend on the presence or specific implementation of any private method when creating your own classes and methods.

Reserved and Optimized Selectors

The following selectors are reserved for the sole use of the GemStone Smalltalk kernel classes. Those selectors are:

```
ifTrue:              whileTrue:             timesRepeat:
ifFalse:             isKindOf:              to:do:
ifTrue:ifFalse:      _isInteger             to:by:do:
ifFalse:ifTrue:      _isSmallInteger        ==
untilFalse           _isSymbol              ~~
untilTrue            _or:                   _class
whileFalse:          _and:                  _trapOop
```

Redefining a reserved selector has no effect; the same primitive method is called and your redefinition is ignored.

In addition, the following methods are optimized in the class SmallInteger:

```
+    -     *     >=     =     >     <     <=     ~=
```

You can redefine the optimized methods above in your application classes, but redefinitions in the class SmallInteger are ignored.

Saving Class and Method Definitions in Files

It's often useful to store the GemStone Smalltalk source code in text files. Such files make it easy to:

- transport your code to other GemStone systems,
- perform global edits and recompilations,
- produce paper copies of your work, and
- recover code that would otherwise be lost if you are unable to commit.

To save GemStone code in a file, use any of the GemStone browser's **File Out** menu items. To read and compile a saved file, use any of the **Gs-File in** or **GS-File it in** menu items.

Saved GemStone files are written as sequences of Topaz commands. Example 3.2 shows a class definition in Topaz format:

Example 3.2

```
doit
Object subclass: 'Address'
  instVarNames: #( street zip)
  classVars: #()
  classInstVars: #()
  poolDictionaries: #[]
  inDictionary: UserGlobals
  constraints: #[ ]
  instancesInvariant: false
  isModifiable: false

%

! Remove existing behavior from Address
doit
Address removeAllMethods.
Address class removeAllMethods.
%
! ------------------- Class methods for Address
! ------------------- Instance methods for Address
category: 'Accessing'
method: Address
street

   "Return the value of the instance variable 'street'."
   ^street
%
category: 'Updating'
method: Address
street: newValue

   "Modify the value of the instance variable 'street'."
   street := newValue
%
category: 'Accessing'
method: Address
zip
```

```
   "Return the value of the instance variable 'zip'."
   ^zip
%
category: 'Updating'
method: Address
zip: newValue

   "Modify the value of the instance variable 'zip'."
   zip := newValue
%
```

GemStone's filing out and filing in facilities are intended mainly for saving and restoring classes and methods without manual intervention. If this is all you want to do, then you don't need to understand the Topaz commands involved. However, it is also possible to create custom files that include commands to commit transactions and to create and manipulate objects other than classes and methods. If you want to perform such tasks, refer to the *Topaz Programming Environment*.

The file-in mechanism cannot execute the full set of Topaz commands; it's limited to the following subset:

```
category:          method
classmethod        method:
classmethod:       printit
commit             removeAllMethods
doit               removeAllClassMethods
```

The GemStone file-in mechanism acknowledges the presence of the following commands by adding notes to the System Transcript, but it does not execute them:

```
display            omit
expectvalue        output
level              remark
limit              status
list               time
```

If GemBuilder encounters any other Topaz commands it stops reading the file and displays an error notifier.

The file-in mechanism does not display execution results, either. Instead, it appends information to the System Transcript about the files it reads and the classes and categories for which it compiles methods.

Handling Errors While Filing In

If one of the modules (run commands or method definitions) that you're filing in contains a GemStone Smalltalk syntax error, GemStone displays a compilation error notifier that contains the erroneous module in a text editor. If you correct the error and then choose **Save**, GemStone recompiles the module and then processes the rest of the file.

In the case of authorization problems, commands that the file-in mechanism doesn't recognize, or other errors, GemStone displays a simple error notifier without an editor and stops processing the file.

Chapter 4

Debugging

In addition to the basic code development tools described in the previous chapter, GemStone also provides GemStone Smalltalk debugging facilities similar to the debugging aids supplied by the client Smalltalk. This chapter describes the important differences between them.

Inspectors
describes how to view and modify the instance variables of server objects

Breakpoints
describes breakpoints, setting breakpoints, and using the Breakpoint Browser

Debugger
describes GemBuilder's enhanced debugger

Stack Traces
describes GbsStackDumper, GemBuilder's enhanced stack dumping facility

4.1 Debugging Overview

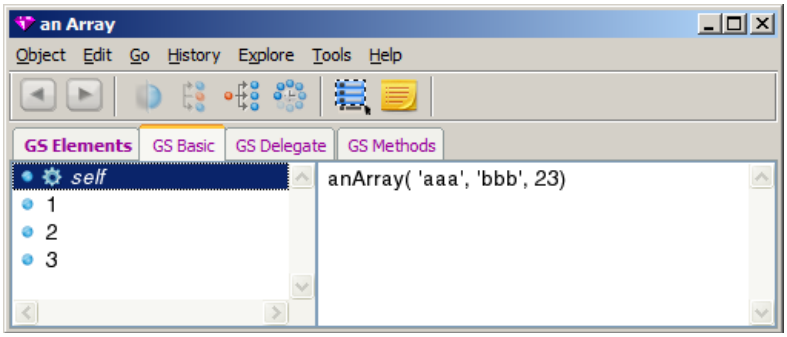
GemBuilder's debugging tools assist you in examining and modifying application objects during execution. These facilities enable you to perform the following operations:

- You can view and alter the instance variables of server objects.
- You can step through execution of a method, examining the values of arguments, temporaries, and instance variables after each step.
- You can set, clear, and examine GemStone Smalltalk breakpoints. When a breakpoint is encountered during normal execution, a debugger opens with which you can interactively explore the contexts in the stack at the time execution halted.
- You can inspect or change the values of arguments, temporaries, and receivers in any context (stack frame) on the virtual machine call stack, then continue execution from the top of the stack. This means that you can find out what the system was doing at the time a breakpoint, or an error interrupted execution.
- You can execute a message expression within the scope of a given context.

4.2 Inspectors

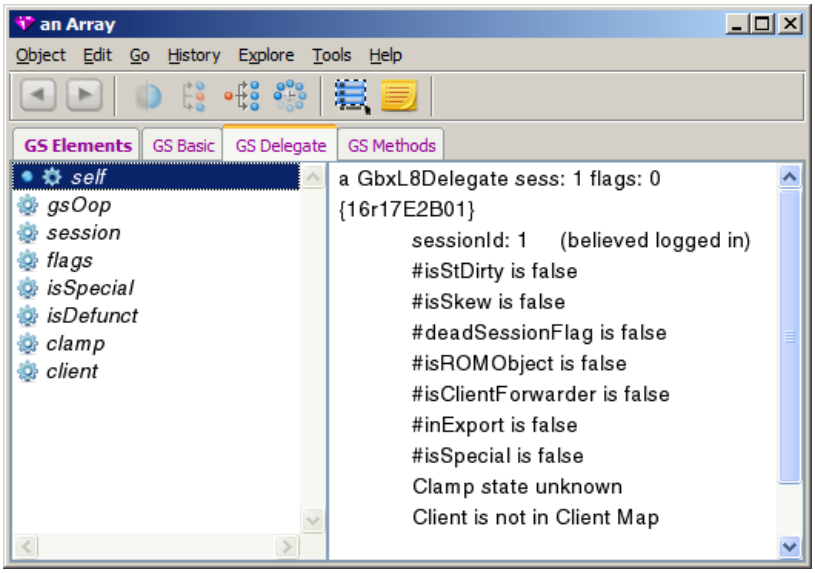
To allow you to examine the values of GemStone server objects and modify them when appropriate, GemBuilder provides inspectors that are similar to the client Smalltalk inspectors. When you select a GemStone Smalltalk expression and execute **GS-Inspect it**, a GemStone inspector opens, as in Figure 4.1.

Figure 4.1 GemStone inspector



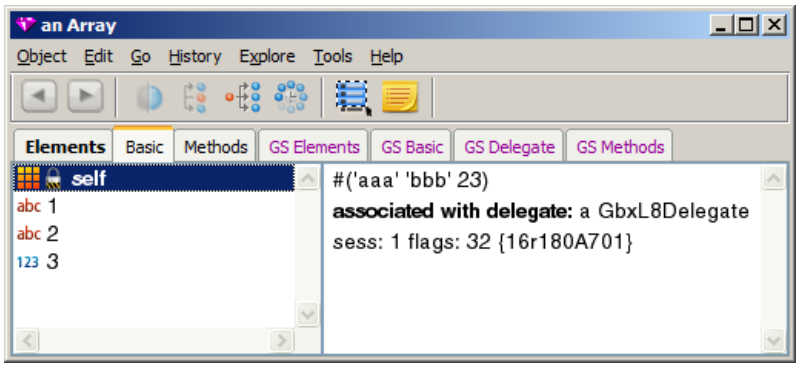
The GemStone inspector provides three or more tabs (depending on the specific object you are inspecting), each with the GS prefix to indicate that these apply to a GemStone server object. The GemStone inspector also allows you to examine the internal state of the object's delegate in GemBuilder, by selecting the tab labeled "GS Delegate".

Figure 4.2 GemStone Inspector GS Delegate tab



When inspecting an object that has a corresponding GemStone server object, the inspector allows you to view both the client object and the server object in the same inspector. Client objects with corresponding server objects includes replicates of server objects, forwarders to server objects, stubs, and objects that are always mapped such as nil, true, false, SmallIntegers, and Characters. When you are inspecting one of these objects, you will see two sets of tabs; one that allows you to examine the client object, and a second similar set of tabs with the GS prefix, to examine the associated GemStone server object.

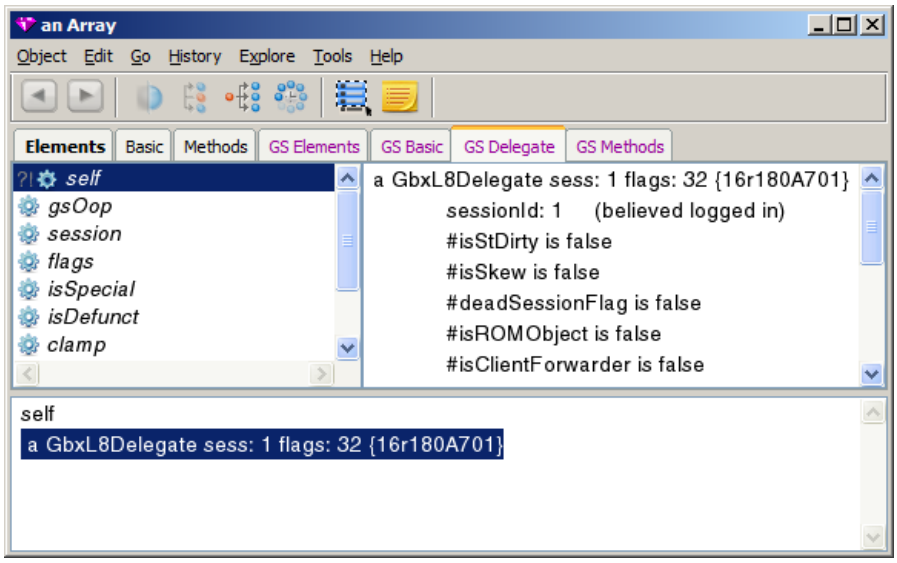
Figure 4.3 Inspector on a Replicate


an Array
Object Edit Go History Explore Tools Help
Elements Basic Methods GS Elements GS Basic GS Delegate GS Methods
self
abc 1
abc 2
123 3
#('aaa' 'bbb' 23)
associated with delegate: a GbxL8Delegate
sess: 1 flags: 32 {16r180A701}

In the evaluation pane of the inspector, or any other pane, the use of “self” is resolved according to the following rules:

- Do it, Print it, Inspect it and Debug it resolve self to the client object
- GS-Do it, GS-Print it, GS-Inspect it, and GS-Debug it resolve self to the GemStone server object
- When the GS Delegate tab is selected, Do it, Print it, Inspect it and Debug it resolve self to the delegate object, as in Figure 4.4.

Figure 4.4 Evaluating “self” in the GS Delegate tab of the Inspector



4.3 Breakpoints

For the purpose of determining exactly where a step will go during debugging, a GemStone Smalltalk method is composed of step points. You can set breakpoints at any step point.

Generally, step points correspond to the message selector and, within the method, message-sends, assignments, and returns of nonatomic objects. However, compiler optimizations may occasionally result in a different, nonintuitive step point, particularly in a loop.

Example 4.1 indicates step points with numbered carets.

Example 4.1

```
includesValue: value
^1

"Return true if the receiver contains an object of the same
value as the argument. Return false otherwise."

| found index size|

found := false.
      ^2
index := 0.
      ^3
size := self size.
     ^5      ^4
[ found not & (index < size)] whileTrue: [
        ^6  ^8       ^7       ^9
      index := index + 1.
            ^11      ^10
      found := value = (self at: index)
            ^14      ^13     ^12
      ].
^found
^15
```

If you use the GemStone debugger (described starting on page 77) to step through this method, the first step takes you to the point where `includesValue:` is about to be sent. Stepping again sends that message and halts the virtual machine at the point where `found` is assigned. Another step sends that message and halts the virtual machine just before the result is assigned to `index`, and so on.

When the GemStone Smalltalk virtual machine encounters an enabled breakpoint during normal execution, GemStone opens the GemStone Debugger. In the Debugger, you can interactively explore the context in which execution halted.

Special considerations apply in setting breakpoints for primitive and special methods.

Breakpoints for Primitive Methods

If you set a breakpoint in a primitive method, the break is encountered only if the primitive fails. Consider the method below:

```
= aString

<primitive: 160>
self _primitiveFailed: #=
```

When this method is invoked, GemStone first executes the machine code in primitive 160. If that code executes successfully, the primitive is said to succeed, and the method returns a value. Because no GemStone Smalltalk code has yet been encountered, the virtual machine has not yet reached the first step point. Only if the primitive fails will the virtual machine execute the message-send at the bottom of the method and thus encounter the breakpoint.

Breakpoints for Optimized Methods

Certain simple methods are optimized by the GemStone Smalltalk compiler in such a way that they contain no step points. Naturally, you cannot set a method breakpoint if there are no step points. A method that performs only one of the following operations has no step points:

- return `true`,
- return `false`,
- return `nil`,
- return `self`,
- return the value of an instance variable,
- assign to an instance variable, or
- return a class or pool variable or a variable defined in a symbol dictionary.

A method that performs some computation and then performs one of the actions listed above contains step points; it is not a simple method for purposes of this discussion.

In addition to the special methods listed above, a handful of specific kernel class methods are specially optimized so that they cannot take breakpoints. Those methods are:

```
ifTrue:              whileTrue:              timesRepeat:
ifFalse:             isKindOf:               to:do:
ifTrue:ifFalse:      _isInteger              to:by:do:
ifFalse:ifTrue:      _isSmallInteger         ==
untilFalse           _isSymbol               ~~
untilTrue            _or:                    _class
whileFalse:          _and:                   _trapOop
```

To set breakpoints, GemBuilder provides the Breakpoint Browser. To debug the resulting stack, GemBuilder enhances the client Smalltalk debugger.

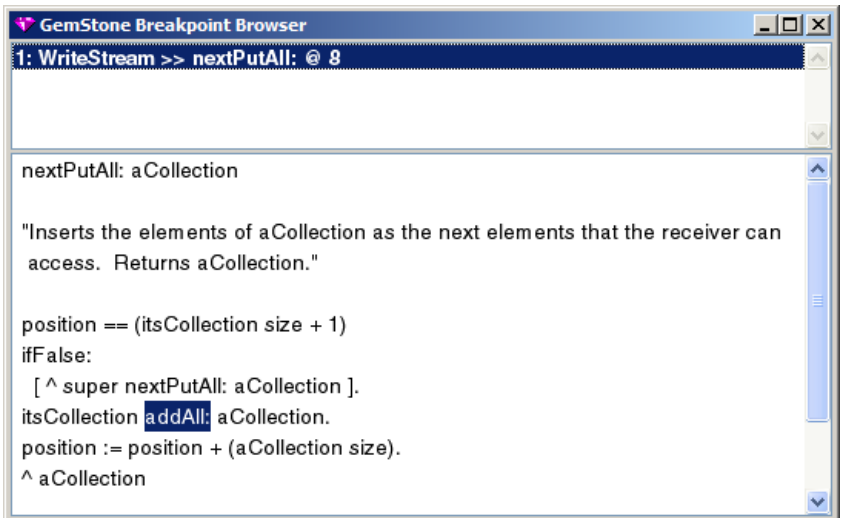
The Breakpoint Browser

You can set breakpoints in the source code pane of any browser, using the **set break** menu item described in Table 3.4 on page 3-56. You can also use the breakpoint browser, which lets you set, clear, and examine breakpoints for all classes and methods.

After you've set a breakpoint, you can use the menu items to disable or re-enable all breakpoints, or just selected ones.

A breakpoint browser has two panes: the list of break points on top, and the source code associated with the selected breakpoint on the bottom. Figure 4.5 shows an example:

Figure 4.5 GemStone Breakpoint Browser with a Breakpoint


GemStone Breakpoint Browser
1: WriteStream >> nextPutAll: @ 8
nextPutAll: aCollection
"Inserts the elements of aCollection as the next elements that the receiver can
access. Returns aCollection."
position == (itsCollection size + 1)
ifFalse:
[^ super nextPutAll: aCollection].
itsCollection addAll: aCollection.
position := position + (aCollection size).
^ aCollection

The Break Pane

The break pane displays a scrollable list of the active breakpoints. The items in the list look like this:

```
1: WriteStream >> nextPutAll: @ 8
```

In this example, a method break is set at step point 8 within the method `nextPutAll:` defined by class WriteStream.

The Source Pane

If you have selected a breakpoint in the break pane, the text area displays the source code for that method. This pane is similar to the GemStone Browser text area, but you cannot recompile an edited method by executing **Save**.

4.4 Debugger

The GemStone Debugger is integrated with the client Smalltalk debugger, allowing you to:

- view GemStone Smalltalk and client Smalltalk contexts together in one stack,

- select a context from among those active on the virtual machine stack,
- examine and modify objects and code within that context, and
- continue execution either normally or in single steps.

When GemStone Smalltalk execution is interrupted, it either directly opens the Debugger, or a notifier that includes a **Debug** button. Selecting the **Debug** button opens the Debugger. A runtime error opens a notifier, while a breakpoint, user interrupt, or an Object >> `pause` opens a debugger.

The Debugger's stack pane displays the active call stack and allows you to choose some context (stack frame) from that stack for manipulation in the window's other panes. Both GemStone server and client contacts are listed. GemStone server contexts begin with "`GS`".

Like other GemBuilder text areas, the debugger source code pane provides commands to execute GemStone Smalltalk.

Colored contexts

You may configure VisualWorks to color the GemStone server contexts, to distinguish them from client contexts. To do this, select the Settings button on the Launcher toolbar, or go to **System > Settings**, and select **Debugger** (under **Tools**). Select the button to edit the Context List Presentation. If no Patterns exist, create a new Pattern for Everything with the Pattern String *. Then, create a new Pattern with the Pattern String **`$L$*`**, and select a color for the GemStone server contexts text. This new Pattern must be above the Pattern String *.

Note that for Windows-based VisualWorks Look and Feel settings, the selection background color may leave selected context text unreadable when colored contexts are enabled.

Disabling the Debugger

In some cases, you may want to disable the GBS debugger. You can disable and enable the debugger using the following expressions:

```
GBSM enableGbsDebugger
GBSM disableGbsDebugger
```

Disabling the GBS debugger restores the base VisualWorks debugger.

4.5 Stack Traces

In some situations it is easier to extract complete stack traces for later analysis, rather than debugging interactively. In addition, you may need a stack trace to provide to GemStone Technical Support. GemBuilder includes facilities to dump the complete stack, with more information than provided in the standard stack, including information on GemStone server contexts and "glue" contexts.

To extract a complete stack, execute

```
GbsStackDumper dumpAllProcessStacks
```

In response, all processes in the image write their complete contexts to a file named `stacksAt`*x*`.txt` in the current working directory, where *x* is a 10-digit number derived from a time stamp.

To dump the stacks to a particular file location:

```
GbsStackDumper dumpAllProcessStacksToFileNamed: aString
```

These methods do not require the debugger, and can be used in runtime applications.

Chapter 5

Sharing Objects

This chapter describes how GemBuilder shares objects with the GemStone/S object repository.

Which Objects to Share?
is an overview of the process of determining how to make good use of GemBuilder's resources, and introducing forwarders, and replicates, and stubs.

Class Mapping
explains how classes are defined and how forwarders, stubs, and replicates depend on them.

Forwarders
explains how to use forwarders to store all an object's state and behavior in one object space.

Replicates
explains replicating GemStone server objects in client Smalltalk, or vice-versa; describes the processes of propagating changes to keep objects synchronized; presents various mechanisms to minimize performance costs; presents further details.

Precedence of Replication Mechanisms
discusses the various ways replication mechanisms interact, and describes

how to determine whether an application object becomes a forwarder, stub, or replicate.

Converting Between Forms
lists protocol for converting from and to delegates, forwarders, stubs, replicates, and unshared client objects.

5.1 Which Objects to Share?

Working with your client Smalltalk, you had one execution engine—the virtual machine—acting on one object space—your image. Now that you've installed GemBuilder, you have *two* execution engines and *two* object spaces, one of which is a full-fledged object repository for multiuser concurrent access, with transaction control, security protections, backups and logging.

What's the best way to make use of these new resources?

Objects represent both state and behavior. Therefore, you have two basic decisions:

- Which state should reside on the client, which on the server, and which in both object spaces?
- Which behavior should reside on the client, which on the server, and which in both object spaces?

Ultimately, the answer is dictated by the unique logic of your specific problem and solution, but these common patterns emerge:

Client presents user interface only; state (domain objects) and application logic reside on server; server executes all but user interface code. A web-based application that uses the client merely to manage the browser needs little functionality on the client, and what it does need is cleanly delimited.

State resides on both client and server; client manages most execution; server is used mainly as a database. A Department of Motor Vehicles could use a repository of driver and vehicle information, properly defined, for a bevy of fairly straightforward client applications to manage driver's licenses, parking permits, commercial licenses, hauling permits, taxation, and fines.

Execution occurs, and therefore state resides, on both client and server. At specified intervals, clients of a nationwide ticket-booking network download the current state of specific theaters on specific dates. Clients book seats and update their local copies of theaters until they next connect to the repository. To resolve conflicts, server and client engage in a complex negotiation.

For these and other solutions, GemBuilder provides several kinds of client- and server-side objects, and a mechanism—*a connector*—for describing the association between pairs of root objects across the two object spaces.

Three kinds of objects help a GemBuilder client and a GemStone server repository share state and execution: forwarders, stubs, and replicates.

Forwarder — is a *proxy:* a simple object that knows only which object in the other space it is associated with. It responds to a message by passing it to its associated master object in the other object space, where state is stored and execution occurs remotely. Forwarders can be on the client, for server master objects, or on the server for client master objects.

Replicate — is an object associated with a particular object in the other object space. The replicate copies some or all of the other object's state, which it synchronizes at appropriate times; it implements all messages it expects to receive; it executes locally.

Stub — is a proxy that responds to a message by becoming a replicate of its counterpart object, then executing the message locally. Stubbing is a way to minimize memory use and network traffic by bringing only what is needed when it is needed.

Connector — associates a root client object with a root server object, typically resolving objects by name, although there are other ways. When connected, they synchronize data or pass messages in either direction or take no action at all, as specified. For more information on connectors, see Chapter 6.

Whatever combination of these elements your application requires, subsystems of objects will probably reside on both the client and the server. Some subset of these subsystems will need state or behavior on both sides: some objects will be shared.

5.2 Class Mapping

Before GemBuilder can replicate an object, it must know the respective structures of client and repository object and the mapping between them. Although not strictly necessary for forwarders, this knowledge improves forwarding performance, saving GemBuilder an extra network round-trip during the initial connection.

GemBuilder uses class definitions to determine object structure. To replicate an object:

- both client and server must define the class, and
- the two classes must be mapped using a *class connector*.

GemBuilder uses this mapping for all replication, whether at login or later.

Unlike connectors for replicates or forwarders, class connectors by default do not update at connect time. If class definitions differ on the client and the server, it is usually for a good reason; you probably don't want to update GemStone with the client Smalltalk class definition, or vice-versa.

GemBuilder predefines special connectors, called *fast connectors*, for the GemStone kernel classes. For more information about fast connectors, see "Connecting by Identity: Fast Connectors" on page 129.

If there is no connector for a class, and a mapping for that class is required, GemBuilder will attempt to map the client and server classes with the same name. By default, it will also create a connector for those classes. If the configuration parameter generateClassConnectors is false, GemBuilder will still map the classes by name, but will not create a connector. The difference is that without a connector, the mapping only lasts until the session logs out, and any other sessions logged in will not have that mapping. If a connector is created, it is associated with the session parameters object, and any session logged in using that session parameters object will have that class mapping created at login time.

Automatic Class Generation and Mapping

By default, GemBuilder generates class definitions and connectors automatically as necessary. If GemBuilder requires the GemStone server to replicate an instance of a client class that is not already defined on the server, then at the first access, GemBuilder generates a server class having the same schema and position in the hierarchy, and a class connector connecting it to the appropriate client class. Conversely, if the client must replicate an instance of a GemStone class that is not already defined in client Smalltalk, GemBuilder generates the client Smalltalk class and the appropriate class connector. If superclasses are also undefined, GemBuilder generates the complete superclass hierarchy, as necessary.

You can control automatic class generation with the configuration parameters **`generateServerClasses`** and **`generateClientClasses`** (described starting on page 227). These settings are global to your image.

- If you disable automatic generation of GemStone classes by setting **`generateServerClasses`** to `false`, situations that would otherwise

generate a server class instead raise the exception `GbsClassGenerationError`.

- If you disable automatic generation of client Smalltalk classes by setting **`generateClientClasses`** to `false`, situations that would otherwise generate a client Smalltalk class instead raise the exception `GbsClassGenerationError`.
- You can disable class connector generation by setting **`generateClassConnectors`** to `false`. If you do so, GemBuilder generates classes, but not their corresponding connectors.

GemBuilder deposits automatically generated GemStone server classes in the GemStone symbol dictionary UserClasses, which it creates if necessary. Automatically generated client Smalltalk classes are deposited in the current package.

Automatic class generation is primarily useful as a development-time convenience. In an application runtime environment, we recommend having all necessary classes predefined in both object spaces, and having a connector defined for each class before logging in. This can improve performance by avoiding unnecessary work when the class is first accessed.

Schema Mapping

By default, when you map a client class to a GemStone server class, GemBuilder automatically maps all instance variables whose names match, regardless of the order in which they are stored. (You can change this default mapping to accommodate nonstandard situations.)

If you later change either of the mapped class definitions, GemBuilder automatically remaps identically named instance variables.

Behavior Mapping

When GemBuilder generates classes automatically, it only copies the definition of the class, not the methods of the class.

Replicated instances depend on methods implemented in the object space in which they execute. During development, it may be simplest to use GemBuilder's programming conveniences to implement the same behavior in both spaces. For reliability and ease of maintenance, however, some decide to remove unnecessary duplication from production systems and to define behavior only where it executes.

Mapping and Class Versions

Unlike the client Smalltalk language, GemStone Smalltalk defines *class versions:* when you change a class definition, you make a new version of the class, which is added to an associated class history. (For details, see the chapter entitled "Class Versions and Instance Migration" in the *GemStone Programming Guide*.)

If you decide to update one class definition with the other, the result depends on the direction of the update:

- Updating a client Smalltalk class from a GemStone server class regenerates the client class and recompiles its methods.
- Updating a GemStone server class from a client Smalltalk class creates a new version of the class.

NOTE

A class connector connects to a specific GemStone class version, the version that was in effect when the connector was connected. Instances of a given class version are not affected by a connector connected to another class version.

5.3 Forwarders

The simplest way to share objects is with f*orwarders,* simple objects that know just one thing: to whom to forward a message. A forwarder is a proxy that responds to messages by forwarding them to its counterpart in the other object space.

Forwarders are particularly useful for large collections, generally resident on the GemStone server, whose size makes them expensive to replicate and cumbersome to handle in a client image.

Forwarders are of two kinds:

- The most common kind of forwarder is a *forwarder to the server:* a client Smalltalk object, an instance of GbsFowarder, that knows only which GemStone server object it represents. It responds to all messages by passing them to the appropriate server object, where its associated state resides and behavior is implemented. (For historical reasons, this is the kind of forwarder usually meant when a discussion merely says "forwarder." This kind of forwarder is also called a server forwarder.)
- A *forwarder to the client* is a GemStone server object that knows only which client Smalltalk object it represents. It responds to all messages by passing

them to its associated client Smalltalk object, where state resides and behavior is implemented.

You can create forwarders in several ways:

- Create a connector with a postconnect action of #forwarder or #clientForwarder. For example, connect the server global variable BigDictionary as a forwarder to the server so that it isn't replicated in the client.
- Specify that a given instance variable must always appear on the client as a forwarder to the server (using a replication specification, discussed starting on page 99). For example, a client class might implement a specification that declares the instance variable inventory as a forwarder to the server.
- Prefix `fw` to a method name to return a forwarder from any message-send to the server. For example, to return a forwarder from a GemStone server name lookup, send the GbsSession `fwat:` or `fwat:ifAbsent:` instead of `at:` or `at:ifAbsent:`.
- Override all these by implementing a class method `instancesAreForwarders` to return `true`, and all instances of that class and its subclasses will be forwarders to the server. Subclasses of GbsServerClass already respond `true` to this message; GbsServerClass is an abstract class, and all instances that inherit from it become forwarders to the server. When sent to a class that inherits from GbsServerClass, the instance creation methods `new` and `new:` create a new instance of the class on the server and return a forwarder to that instance.

Sending Messages

On the client, when a forwarder to the server receives a message, it sends the message to its counterpart on the GemStone server—presumably an instance that can respond meaningfully. The target server object's response is then returned to the forwarder on the client, which then returns the result.

When a forwarder to the client receives a message on the server, it forwards the message to the full-fledged client object to which it is connected. This object's response is returned to the client forwarder, which returns the result represented as a server object.

Arguments

Before a message is forwarded to the GemStone server, arguments are translated to server objects. As a message is forwarded to the client, its arguments are translated to client Smalltalk objects.

When an argument is a block of executable code, special care is required: for details, see “Replicating Client Smalltalk BlockClosures” on page 109.

Results

The result of a message to a client forwarder is a GemStone Smalltalk object in the GemStone server.

The result of a message to a server forwarder is the client Smalltalk object connected to the server object returned by GemStone—usually a replicate, although a forwarder might be desirable under certain circumstances.

To ensure a forwarder result, prefix the message to the forwarder with the characters `fw`. For example:

- `aForwarder at: 1` returns a *replicate* of the object at index 1.
- `aForwarder fwat: 1` returns a *forwarder* to the object at index 1.

Defunct Forwarders

A forwarder contains no state or behavior in one object space, relying on the existence of a valid instance in the other. When a session logs out of the server, communication between the two spaces is interrupted. Forwarders that relied on objects in that session can no longer function properly. If they receive a message, GemBuilder raises an error complaining of either an invalid session identifier or a defunct forwarder.

You cannot proceed from either of these errors; an operation that encounters one must restart (presumably after determining the cause and resolving the problem).

GemBuilder cannot safely assume that a given server object will retain the same object identifier (OOP) from one session to the next. Therefore, you can’t fix a defunct forwarder error simply by logging back in.

(If a connector has been defined for that object or for its root, then logging back in will indeed fix the error, because logging back in will connect the variables. But in that case, it’s the connector, not the forwarder, that repairs damaged communications.)

Consider the following forwarder for the global BigDictionary:

Example 5.1

```
conn := GbsNameConnector
      stName: #BigDictionary
      gsName: #BigDictionary.
conn beForwarderOnConnect.
GBSM addGlobalConnector: conn
```

When a GemBuilder session logs into the GemStone server, BigDictionary becomes a valid forwarder to the current server BigDictionary. But when no session is logged into the server, sending a message to BigDictionary results in a defunct forwarder error.

GemBuilder's configuration parameter **connectorNilling**, when true, assigns each connector's variables to `nil` on logout. This applies only to session-based name, class variable, or class instance variable connectors that have a postconnect action of `#updateST` or `#forwarder` (See "Connectors" on page 121). This usually prevents defunct stub and forwarder errors, replacing them with `nil doesNotUnderstand` errors.

5.4 Replicates

Sometimes it's undesirable to dispatch a message to the other object space for execution—sometimes local execution is desirable, even necessary, for example, to reduce network traffic. When local state and behavior is required, share objects using replicates instead of forwarders. Replicates are particularly useful for small objects, objects having visual representations, and objects that are accessed often or in computationally intensive ways.

Like a forwarder, a *replicate* is a client Smalltalk object associated with a server object that the replicate represents. Unlike a forwarder, replicates also hold (some) state and implement (some) behavior. Replicates synchronize their state with that of their associated server object.

To do so, GemBuilder must know about the structure of the two objects and the mapping between those structures. GemBuilder manages this mapping on a class basis: each replicate must be an instances of a class whose definition is mapped to the definition of the corresponding class in the server object space. GemBuilder handles many obvious cases automatically, but nonstandard mappings require you to implement certain instance and class methods. Nonstandard mappings are discussed starting on page 92.

Synchronizing State

After a relationship has been established between a client object and a GemStone server object, GemBuilder keeps their states synchronized by propagating changes as necessary.

When an object changes in the server, GemBuilder automatically updates the corresponding client Smalltalk replicate. By default, GemBuilder also detects changes to client Smalltalk replicates and automatically updates the corresponding server object.

The stages and terminology of this synchronization are as follows:

- When an object is modified in the client, leaving its server counterpart out of date, the client object is now referred to as *dirty*.
- When the state of dirty client objects is transferred to their corresponding server objects, this is called *flushing*.
- When a server object is modified in the server, leaving its client counterpart out of date, the server object is now *dirty*. This can occur during execution of server Smalltalk, or at a transaction boundary when changes committed by other sessions become visible to your session.
- When the state of dirty server objects is transferred to their corresponding client objects, this is called *faulting*.

Together, GemBuilder and the GemStone server manage the timing of faulting and flushing.

Faulting

GemBuilder faults objects automatically when required. Faulting is required when a stub receives a message, requesting it to turn itself into a replicate. (see stubbing on page 95)

Faulting may also be required when:

- Connectors connect; this typically occurs at login, the beginning of a GemStone session, but you can connect and disconnect connectors explicitly during the course of a session using either code or the Connector Browser. Faulting may or may not occur upon connection, depending on the post-connect action specified for the connector.

- A server object that has been replicated to the client is modified on the server. This can happen in two cases:

  1. GemStone Smalltalk execution in your session modifies the state of the object. GemStone Smalltalk execution occurs when a forwarder receives a message, or in response to any variant of GbsSession >> evaluate:.
  2. Your session starts, commits, aborts, or continues a transaction—passes a transaction boundary—which refreshes your session's private view of the repository. If the server object has been changed by some other concurrent session, and that change was committed, the object's new state will be visible when your session refreshes its view.

  In both of these cases, the replicate's state is now out of date, and cannot be used until updated by faulting. Depending on the replicate's faultPolicy (see page 98) the new state will either be faulted immediately, or the replicate becomes a stub, and will be faulted the next time it receives a message.

Flushing

GemBuilder flushes dirty client objects to the GemStone server at transaction boundaries, immediately before any GemStone Smalltalk execution, or before faulting a stub.

Flushing is not the same as committing. When GemBuilder flushes an object, the change becomes part of the session's private view of the GemStone repository, but it doesn't become part of the shared repository until your session commits—only then are your changes accessible to other users.

For GemBuilder to flush a changed object to the server, that object must be **marked dirty**, that is, GemBuilder must be made aware that the object has changed. Objects can be marked dirty automatically, depending on the version of client Smalltalk, or you can explicitly mark objects dirty.

Marking Modified Objects Dirty Automatically

By default, GemBuilder uses features of the VisualWorks 7.x Object Engine to detect modifications to replicates on the client so that modified replicates can be automatically marked dirty. This mechanism is fast, reliable, and does not affect client objects that are not replicates. Thus, when using VisualWorks 7.x we recommend always using automatic dirty-marking. Automatic dirty-marking is enabled by default.

To disable automatic dirty-marking, execute:

```
GbsConfiguration current autoMarkDirty: false
```

or use the Settings Tool to turn off the configuration parameter `autoMarkDirty`. It is enabled or disabled globally for the client; you cannot enable automatic dirty-marking for only some classes or objects in the client virtual machine. If you disable automatic dirty-marking, your application must manually mark modified client replicates dirty as described in the next section.

Marking Modified Objects Dirty Manually

Generally, we recommend you use the automatic mechanisms. You can instead, if you wish, mark objects dirty explicitly in your code. The automatic mechanism is faster and much more reliable—if you miss even one place where a shared object is modified, your application will misbehave.

To manually mark a replicate dirty, send `markDirty` to the replicate immediately after each time your application modifies it. If a replicate is modified on the client but not marked dirty, the modification will be lost eventually. The object could be overwritten with its GemStone server state after the application has executed code on the server, or at the next transaction boundary. Even if the client object is never overwritten, the modification will never be sent to the server.

Minimizing Replication Cost

Replicating the full state of a large and complex collection can demand too much memory or network bandwidth. Optimize your application by controlling the degree and timing of replication; GemBuilder provides three ways to help:

Instance Variable Mapping — Modify the default class map to specify how *widely* through each object to replicate—which instance variables to connect and which to prune as never being of interest in the other object space. You can also specify the details of an association between two classes whose structures do not match.

Stubbing — Specify how *deeply* through the network to replicate, how many layers of references to follow when faulting occurs.

Replication Specifications — Another way to specify how *widely or deeply* through each object to replicate—of a class's mapped instance variables, which to replicate and which to stub.

Instance Variable Mapping

As discussed in "Class Mapping" on page 83, before GemBuilder can replicate objects, it must know their respective structures and the mapping between them. By default GemBuilder maps instance variables by name. You can override this

default either by suppressing the copying of certain instance variables, or by explicitly specifying a mapping between nonmatching names.

Suppressing Instance Variables

Some client Smalltalk objects must define instance variables that are relevant only in the client environment—for example, a reference to a window object. Such data is transient and doesn't need to be visible to the GemStone server. Situations can also arise in which the server class defines instance variables that a given application will never need; many applications can share repository objects without necessarily sharing the same concerns. Mapping allows your application to prune parts of an object.

Suppress the replication of an individual instance variable simply by omitting its name from its counterpart's class definition:

- If a client object contains a named instance variable that does not exist in its GemStone server counterpart, the value of that variable is not replicated in the server. When GemBuilder faults the server object into the client, the client's suppressed instance variable remains unchanged.
- Likewise, if a server object contains a named instance variable that does not exist in its client counterpart, the value of that variable is not replicated in the client. When GemBuilder flushes the object into the server, the server object's suppressed instance variable remains unchanged.

You can also suppress instance variable mappings by implementing the client class method `instVarMap`. Example 5.2 shows a simple implementation:

Example 5.2

```
TestObject class>>instVarMap
      ^super instVarMap ,
            #(    (nil gsName)
                  (stName nil) )
```

The first component of the return value, a call to `super instVarMap`, ensures that all instance variable mappings established in superclasses remain in effect.

Appended to the inherited instance variable map, an array contains the pairs of instance variable names to map. The first pair (`nil gsName`) specifies that the server instance variable `gsName` will never be replicated in the client. The second pair (`stName nil`) specifies that the client instance variable `stName` will never be replicated in the server.

Nonmatching Names

You can also specify an explicit instance variable mapping between the server and the client:

- to map two instance variables whose names don't match, or
- to prevent the mapping of two instance variables whose names *do* match.

In this way your application can accommodate differing schemas.

To specify nonstandard instance variable mappings, use the same class method `instVarMap`, as in Example 5.3:

Example 5.3

```
TestObject class>>instVarMap
      ^super instVarMap ,
            #(    (stName  gsName) )
```

Appended to the inherited instance variable map, a single pair declares that the instance variable `stName` in the client maps to the instance variable `gsName` in GemStone.

One implementation can both prune irrelevancy and accommodate differing schemas, as the instance variable mapping for the class Book shows in Example 5.4:

Example 5.4

```
Book class>>instVarMap
      ^super instVarMap ,
            #(    (title title)
                  (author author)
                  (nil pages)
                  (publisher nil)
                  (copyright publicationDate) )
```

The first two pairs of instance variables change nothing: they explicitly state what would happen without this method, but are included for completeness.

`(nil pages)` specifies that the client application does not need to know a books page count and therefore this server-side instance variable is not replicated in the client.

`(publisher nil)` specifies that the client application needs (and presumably assigns) the instance variable `publisher`, which is never replicated in the server.

`(copyright publicationDate)` maps the client class Book's instance variable `copyright` to the server class Book's instance variable `publicationDate`.

Stubbing

Often an application has need of certain instance variables, but not all at once. For example, it's impractical to replicate the entire hierarchy of BigDictionary at login: users will experience unacceptable network delays, and the client Smalltalk image can't handle data sets as large as the GemStone server can. Furthermore, it's unnecessary: only a small number of objects will be needed for the current task. To help prevent this kind of over-replication, GemBuilder provides stubs.

A *stub*, like a forwarder, is also a proxy associated with a server object. Unlike a forwarder, however, when a stub receives a message, it does not send the message across to the other object space. Instead, it faults is server counterpart into the client image. The client Smalltalk replicate then responds to the message.

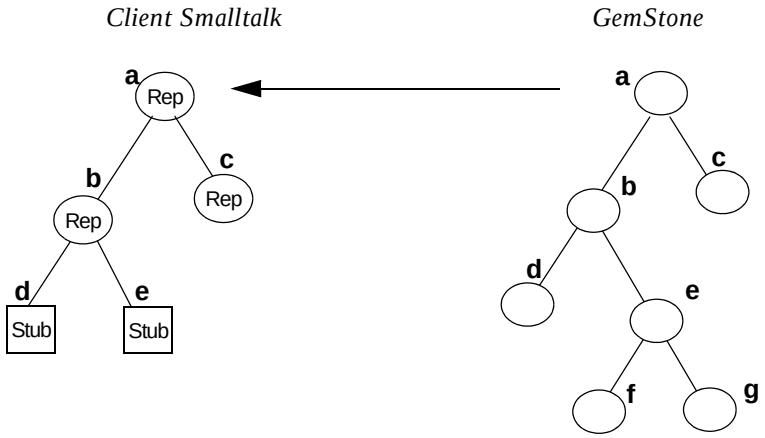
When GemBuilder faults automatically, it replicates the object hierarchy to a certain level, then creates stubs for objects on the next level deeper than that. The number of levels that are replicated each time is the *fault level.*

A fault level of 1 follows an object's immediate references and faults those in. A fault level of 2 follows one more layer of references and replicates those objects, too. Figure 5.1 illustrates an application with a fault level of 2.

Faulting at Login

At login, the connectors connect, and objects **a**, **b**, and **c** are replicated; objects **d** and **e** are stubbed; objects **f** and **g** are ignored.

Figure 5.1 Two-level Fault of an Object



Faulting in Response to a Message

When object **e**, a stub, receives a message, it faults in a replicate of its counterpart GemStone server object.

A stub faults in a replicate in response to a message. Therefore, direct references to instance variables can cause problems. Direct access is not a message-send; the stub will not fault in its replicate, because it receives no message; neither can it supply the requested value. To avoid this problem, use accessor methods to get or set instance variables.

The following sequence demonstrates the problem. The object starts as a replicate in client Smalltalk:

```
myVar := 'abc'.
```

Next, it's stubbed. There are several ways it could become a stub; for this example, we'll assume that it became a stub by locking (locking an object stubs it).

```
self mySession writeLock: self.
```

The object, now a stub, has no knowledge of the replicate's instance variables. Therefore, executing the code below later in the same method causes an error:

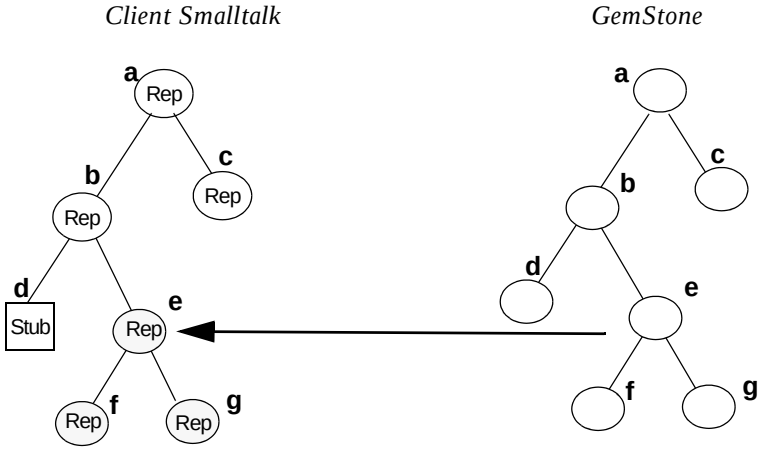
```
myVar := 'bcd'
```

Using an accessor method, on the other hand, causes the stub to be faulted in and yields the correct result:

```
self myVar: 'bcd'
```

e is now a replicate, as shown in Figure 5.2. The new replicate responds to the message.

Figure 5.2 A Stub Responds to a Message



Again, two levels are replicated, object **e** and its immediate instance variable: a fault level is a global parameter.

Faulting in Changes From Other Sessions

Now, suppose another session commits a change to **b**?

Each session maintains its own view of the GemStone object server's shared object repository. The session's private view can be changed by the client application when it adds, removes, or modifies objects—that is, you can see your own changes to the repository—or the Gem can change your view at transaction boundaries or after a session has executed GemStone Smalltalk.

A Gem maintains a list of repository objects that have changed and notifies GemBuilder of any changes to objects it has replicated. If it finds any changed counterparts, it updates the client object with the new GemStone value.

Replicates and stubs respond to the message `faultPolicy`. The default implementation returns the value of GemBuilder's configuration parameter `defaultFaultPolicy`: either `#lazy` or `#immediate`.

- A *lazy* fault policy means that, when GemBuilder detects a change in a repository object, it turns the client counterpart from a replicate into a stub. The object will remain a stub until it next receives a message.
- An *immediate* fault policy means that, when GemBuilder detects a change in a repository object, it updates the replicate immediately.

If another session commits a change to **b**, and **b**'s fault policy is lazy, **b** becomes a stub. If **b**'s fault policy is immediate, **b** is updated.

The default fault policy is lazy, to minimize network traffic. For more information, see the description of `defaultFaultPolicy` in the Settings Browser. For examples, browse implementors of `faultPolicy` in the GemBuilder image.

Overriding Defaults

Because linked sessions can access the gem with lower latency, GemBuilder ships with `faultLevelLnk` set to 2 and `faultLevelRpc` set to 4. In this way, linked sessions replicate less at login, faulting in objects as they are needed.

- You can override these defaults for specific instance variables of specific replicates.
- You can also stub or replicate certain objects explicitly.

To specify fault levels for all instance variables, implement a class method `replicationSpec` for the client class. Replication specifications are versatile mechanisms described starting on page 99.

To cause a replicate to become a stub, send it the message `stubYourself`. This can be useful for controlling the amount of memory required by the client Smalltalk image. Explicit control of stubs is discussed in "Optimizing Space Management" on page 207.

Sometimes stubbing is not desirable, either for performance reasons or for correctness. For example, primitives cannot accept stubs as arguments if the primitive accesses the instance variables of the argument. If your application uses an object as an argument to a primitive, you must either prevent that object from ever becoming a stub, or ensure that it is replicated before the primitive is executed.

To cause a stub to become a replicate, send it the message `fault`. Stubs respond to this message by replicating; replicates return `self`. The message `faultToLevel:` allows you to fault in several levels at once, as specified.

To prevent a replicate from ever being a stub, configure it as a replicate at login and set its `faultPolicy` to `#immediate`.

Defunct Stubs

Faulting in a stub relies on the existence of a valid GemStone server object to replicate or forward to. If an object is stubbed and the session logs out, a message to that stub raises an error complaining that it is *defunct.* For example, suppose MyGlobal is modified in the server, thereby stubbing it in your client session. If the session logs out before MyGlobal is faulted back in, the client Smalltalk dictionary contains a defunct stub.

Because GemBuilder cannot safely assume that a given object will retain the same object identifier from one session to the next, it cannot simply fix the problem at next login. That's the job of a connector: to reestablish at login the stub's relationship to GemStone. A connector can do so either directly, by connecting the stub itself, or transitively, by connecting some object that refers to the stub.

If you've defined a connector for MyGlobal, logging back into GemStone reconnects it.

Now, suppose an instance variable of MyGlobal becomes a stub shortly before a session logs out. Sending a message to this variable will produce a defunct stub error. At next login, MyGlobal's connector will fault in the variable. You can then retry the message, but only by means of a message sent to MyGlobal (or another connected object). If the application is maintaining a direct reference to the previous defunct stub, the error will persist.

NOTE

You cannot proceed from a defunct stub error. After you've encountered this error, determined the cause, and corrected the problem, you must restart the client Smalltalk operation that encountered the defunct stub.

Replication Specifications

By default, when GemBuilder replicates an instance of a connected class, it replicates all that class's instance variables to the session's specified fault level. You can further refine faulting by class, however, with specific instructions for individual instance variables.

Each class replicates according to a replication specification (hereafter referred to as a *replication spec).* The replication spec allows you to fault in specified instance

variables as forwarders, stubs, or replicates that will in turn replicate their instance variables to a specified level.

By default, a class inherits its replication spec from its superclass. If you haven't changed any of the replication specs in an inheritance chain, then the inherited behavior is to replicate all instance variables as specified by the configuration parameters `faultLevelLnk` and `faultLevelRpc`.

To modify a class's replication behavior in precise ways, implement the class method `replicationSpec`. For example, suppose you want class Employee's address instance variable always to fault in as a forwarder:

Example 5.5

```
Employee >> replicationSpec
      ^ super replicationSpec ,
      #(    ( address forwarder )).
```

To ensure that replication specs established in superclasses remain in effect, Example 5.5 appends its implementation to the result of:

```
super replicationSpec
```

Appended to the inherited replication spec are nested arrays, each of which pairs an instance variable with an expression specifying its treatment at faulting:

(*instVar* *whenFaulted*)

instVar can be either:

- the client-side name of an instance variable, or
- the reserved identifier `indexable_part`, specifying an object's unnamed indexable instance variables, such as the elements of a collection.

whenFaulted is one of:

`stub` — faults in the instance variable as a stub.

`forwarder` — faults in the instance variable as a forwarder to the server.

`min` *n* — faults in the instance variable and its referents as replicates to a minimum of *n* levels. `min 0` = replicate.

`max` *m* — faults in the instance variable and its referents as replicates to a maximum of *m* levels. `max 0` = stub.

`replicate` — faults in the instance variable as a replicate whose behavior will be subject to the configuration parameters `faultlevelRpc` and `faultLevelLnk`, relative to the root object being faulted.

By default, an instance variable's behavior is `replicate`; your application needn't specify replicates unless to restore behavior overridden in a superclass.

Example 5.6

```
TestObject class>>replicationSpec
^super replicationSpec ,
      #(    (instVar1 stub)
            (instVar2 forwarder)
            (instVar3 max 0)
            (instVar4 min 0)
            (instVar5 max 2)
            (instVar6 min 2)
            (instVar7 replicate)
            (indexble_part min 1) )
```

Replication Specifications and Class Versions

As explained in "Mapping and Class Versions" on page 86, client Smalltalk classes connect not simply to GemStone Smalltalk classes, but to specific server class versions. A class connector connects to only one server class version.

A replication spec, therefore, affects only client instances connected to instances of the correct GemStone class version.

Suppose, for example, that you define and redefine class X in the server until its class history lists three versions. Your client Smalltalk class is connected to Version 2. Class X's replication spec will affect server instances of Class X, Version 2. If the server contains instances of Class X, Versions 1 or 3, the replication spec will not affect them.

Multiple Replication Specifications

It's not always possible to define one replication spec that works well for all operations in an application. Some queries or windows may require a different object profile than others in the same application and session; a replication spec crafted to optimize one set of operations can make others inefficient.

By default, the message `replicationSpec` returns the default replication spec. Change this by sending the message `replicationSpecSet:`

#someRepSpecSelector to an instance of GbsSession. With this message, you can specify multiple replication specs, selecting one dynamically according to circumstances. The following procedure shows how:

Step 1. Decide on a new name, such as `replicationSpec2`.

Step 2. Implement `Object class >> replicationSpec2` to return `self replicationSpec`.

Step 3. Reimplement `replicationSpec2` as appropriate in those application classes that need it.

Step 4. Immediately before your application performs the query or screen fetch or other operation that requires the second replication spec, send `replicationSpecSet:` *#replicationSpec2* to the current GbsSession instance.

For example, suppose your application has a class Employee, with instance variables `firstName`, `lastName`, and `address`. `address` contains an instance of class Address. The application has one screen that displays the names from a list of employees, and another screen that displays the zip codes from a list of employee addresses. Here's how to replicate only what's needed:

Step 1. Define a new replication spec with the selector `empNamesRepSpec`.

Step 2. Implement `Object class >> empNamesRepSpec` as:

```
^self replicationSpec.
```

Step 3. Implement `Employee class >> empNamesRepSpec` as:

```
^#((firstName min 1) (lastName min 1) (address stub))
```

Step 4. Define another replication spec with the selector `empZipcodeRepSpec`.

Step 5. Implement `Object class >> empZipcodeRepSpec` as:

```
^self replicationSpec
```

Step 6. Define `Employee class >> empZipcodeRepSpec` as:

```
^#((firstName stub) (lastName stub) (address min 2))
```

and `Address class >> empZipcodeRepSpec` as:

```
^#((city stub) (state stub) (zip min 1))
```

Step 7. Before opening the employee names screen, send:

```
myGbsSession replicationSpecSet: #empNamesRepSpec
```

Restore it to `#replicationSpec` after opening the window.

Step 8. Before opening the zip code window, send:

```
myGbsSession replicationSpecSet: #empZipcodeRepSpec
```

Restore it to `#replicationSpec` after opening the window.

For each window, the procedure above reduces the number of objects retrieved to the minimum required. Other objects fault in as stubs; if subsequent input requires them, they are retrieved transparently.

Managing Interobject Dependencies

Replication specs are ordinarily an optimization mechanism. Some applications, however, require a replication spec to function correctly. If the structural initialization of an object depends on other objects, you must implement replication specs to ensure that, when GemBuilder replicates an object, it also replicates those objects it depends on.

For example, in order to create a Dictionary when replicating it from the server, we need to be able to send `hash` to each key to determine its location in the hash table (hash values aren't necessarily the same in the server as they are in the client). So, if GemStone replicates a Dictionary, it must also replicate the association, and the key in the association. The default implementation for `Dictionary class >> replicationSpec` therefore contains `#(indexable_part min 1)`, and `Association class >> replicationSpec` contains `#(key min 1)`.

This works for Dictionaries with simple keys such as strings, symbols or integers. If an application has dictionaries with complex keys, though, additional replication specs can be required. For example, if you are storing Employees as keys in a dictionary, and you've implemented `=` and `hash` in Employee to consider the `firstName` and `lastName`, then you must ensure that when a dictionary containing Employees is traversed, so are the associations, the employees, and the `firstName` and `lastName`.

You could ensure this by implementing `Employee class >> replicationSpec` to include `#(firstName min 1)` and `#(lastName min 1)`. Or, if you had a special Dictionary class for Employees, you could include `#(indexable_part min 3)` in that dictionary class's replication spec. However, this could cause the entire Employee to be replicated whenever one of these dictionaries is replicated, rather than just the `firstName` and `lastName`.

We recommend that you use the default replication spec `#replicationSpec` as the base replication spec for all classes to reflect interobject dependencies. When defining other replication specs, make sure the default implementation in Object is:

```
^self replicationSpec
```

Ensure that subclass implementations of the new `replicationSpec` method do not stray from the default, so as not to break interobject dependencies.

Precedence of Multiple Replication Specs

It's possible to implement replication specs that appear to contradict each other. Such apparent conflicts are resolved deterministically according to the order in which instance variables appear in a replication spec and the order in which objects are replicated. If a superclass specifies one way of handling an instance variable, and a subclass reimplements `replicationSpec` to handle the same variable in a different way, the last occurrence takes precedence.

For example, suppose the value returned from sending `replicationSpec` to the subclass is:

```
#((name min 1) (name max 2))
```

The last occurrence of the instance variable is `max 2`, and therefore takes precedence.

If subclass implementations of `replicationSpec` always append their results to `super replicationSpec`, the subclass will reliably override the superclass handling of a given instance variable. The recommended approach is:

```
^super replicationSpec, #((name max 2))
```

not:

```
^#((name max 2)), super replicationSpec.
```

Another apparent contradiction can arise between parent and child objects. For example, suppose Employee refers to an Address, which refers to a complex object County. The Employee `replicationSpec` includes `#(address min 5)`, specifying that several levels of the County object are to be replicated. But if Address includes `#(county max 1)`, it modifies Employee's handling of address.

Employee specifies, "Get at least 5 levels of address." Address specifies, "Whatever you do, don't get more than one level of county." The apparent contradiction is resolved by the order in which these specifications are

encountered: because Address is encountered after Employee, Address takes precedence.

If your object network includes cycles, different replication specs could take effect at different times, depending on which object is the replication root at any given time. Given a specific root object, however, it's always possible to determine the exact effect of a set of replication specs.

Customized Flushing and Faulting

You can customize both flushing and faulting to change object structure arbitrarily, if your application requires it. You can even create a class in the server GemStone that maps to a client Smalltalk class with a different format—for example, a format of bytes on the client but pointers in the server.

Modifying Instance Variables During Faulting

You can customize object retrieval by using buffers for the client counterparts of GemStone server objects as they are faulted in. You can then process the contents of these buffers in any manner required.

To provide these buffers, reimplement the class methods:

```
namedValuesBuffer
indexableValuesBuffer
```

To unpack these buffers correctly, reimplement the class methods:

```
namedValues:
indexableValues:
namedValues:indexableValues:
```

By default, `namedValuesBuffer` returns self; new client objects are faulted directly into the named instance variable slots. Override this to supply either a different object of the same type, or an instance of GbsBuffer (a subclass of Array) of the required size.

By default, `indexableValuesBuffer` returns `self`. Override this to return an indexable buffer of the appropriate size.

The buffers you define in these methods are used during faulting. They are subsequently unpacked by the faulted object according to its implementation of the unpacking methods listed above.

Implement the unpacking methods to obtain the desired client representation by performing arbitrary computation on the buffer contents. Use the message

`namedValues:indexableValues:` for cases in which computation must operate on indexable and named values together.

NOTE

The methods `namedValuesBuffer` *and* `namedValues:` *are a pair; so are* `indexableValuesBuffer` *and* `indexableValues:`. *To avoid replication errors, if you override one, you must also override the other.*

You can also override the messages `indexableValueAt:put:` and `namedValueAt:put:` to process the values of the indexable and named slots of the object. For example, class Set might implement the former as:

```
Set >> indexableValueAt: index put: aValue
      self add: aValue
```

The method simply adds the element to the Set rather than assigning it to a specific slot.

NOTE

To avoid generating a "previous flush did not complete" error, if you override `namedValues:` *or* `indexableValues:`*, make sure you do not send messages to any stubs that would require a remote object to be faulted. Doing so causes an error as faulting is attempted while flushing. Adjust the* `replicationSpec` *and* `faultPolicy` *of the object to ensure that stubs won't exist for special flush operations.*

You can override two other messages to control faulting initialization and postprocessing: `preFault` and `postFault`.

Implement `preFault` to initialize a newly created object prior to faulting its named and indexable values.

For example:

```
OrderedCollection >> preFault
  "Initialize <firstIndex> and <lastIndex> prior to
   adding elements."
      self setIndices
```

The method `indexableValueAt:put:` for OrderedCollection has an implementation similar to Set to add the indexable objects. As another example, a specialized type of SortedCollection could use `preFault` to assign the sortBlock so that additions to the collection would be sorted properly during faulting.

Implement `postFault` to do any necessary postprocessing. For example, if the methods used to add to an OrderedCollection also marked the object dirty, the postprocessing could remove dirty-marking: by definition, faulting never results in a dirty object (assuming that the valid state is the state on the):

```
OrderedCollection >> postFault
 "Additions to the OrderedCollection are due to the faulting
  mechanisms and should not result in a dirty object."
      self markNotDirty
```

Modifying Instance Variables During Flushing

To provide an arbitrary mapping of objects from the client to the server you can implement two class methods called `namedValues` and `indexableValues`.

`namedValues`
: Implement this to return a copy of the object being stored or an instance of GbsBuffer sized to match the number of named instance variables in the client object. The store operations then access this buffer for storing in the server.

`indexableValues`
: Implement this to return a list of the indexable instance variables in the client object. The store operations then access this list for storing in the server.

Implementations of `namedValues` must return an object with the appropriate number of named instance variable slots. In Example 5.7, a clone of the positionable stream is returned that increments the `position` instance variable by 1 as needed when mapped into the server:

Example 5.7

```
PositionableStream>>namedValues
      | aClone |
      aClone := self copy.
      aClone instVarAt: 1 put: self contents.
      aClone instVarAt: 2 put: position + 1.
      ^aClone
```

An alternative might return an instance of GbsBuffer (a subclass of Array) of the appropriate size. (A special buffer class is necessary to distinguish between trying to store an array and trying to store the named values of an object residing in a buffer.)

The default implementation of `namedValues` is to return `self`. In this case, the instance variables are processed directly from the object being stored, eliminating the need for a temporary array.

Implementations of `indexableValues` must return an indexable collection containing a sequential list of the elements in the collection. In Example 5.8, for class Set, an Array is returned, because the indexable fields of a Smalltalk set are a sparse list of the actual elements.

Example 5.8

```
Set>>indexableValues
      | values index |
      values := Array new: self size.
      index := 1.
      self elementsDo: [:each |
            values at: index put: each.
            index := index + 1].
      ^values
```

The default implementation of `indexableValues` is to return `self`. In this case, the indexable slots are processed directly from the object being stored, eliminating the need for a temporary array.

You can also override the messages `indexableValueAt:` and `namedValueAt:` to return processed values rather than the actual values in the indexable and named slots of the object. For example, OrderedCollection might implement `indexableValueAt:` as:

```
OrderedCollection>indexableValueAt: index
      ^self at: index
```

This lets OrderedCollection control for the fact that its underlying indexable slots are being managed by the `firstIndex` and `lastIndex` instance variables—that is, the first actual indexable slot of the object may not necessarily be the first logical element.

In conjunction with these two methods, you might need to reimplement the messages `indexableSize` and `namedSize` as well. For example, to match the implementation of `indexableValueAt:` above, OrderedCollection would have to implement `indexableSize` as shown below; otherwise, the object storage

mechanisms would try to iterate over the entire list of indexable slots rather than those controlled by `firstIndex` and `lastIndex`:

```
indexableSize
      ^self size
```

Mapping Classes With Different Formats

You can create a class in GemStone that maps to a client Smalltalk class with a different format—for example, a format of bytes on the client but pointers in the repository. To do so, reimplement the class method `gsObjImpl` in the client Smalltalk to return a value specifying the GemStone implementation.

A `gsObjImpl` method must return a SmallInteger representing the GemStone class format. The following formats are valid:

Return	Format
0	pointers
1	bytes
2	nonsequenceable collection

Symbolic names for these values are stored in the pool dictionary SpecialGemStoneObjects.

Limits on Replication

Replicating blocks and collections with instance variables can present special problems, discussed below.

Replicating Client Smalltalk BlockClosures

Forwarders are especially well-suited for managing large collections that reside in the object server. Collections are commonly sent messages that have blocks as arguments. When the collection is represented in client Smalltalk by a forwarder, these argument blocks are replicated in GemStone and executed in the server.

When a GemStone replicate for a client Smalltalk block is needed, GemBuilder sends the block to GemStone Smalltalk for recompilation and execution. If a block is used more than once, GemBuilder saves a reference to the replicated block to avoid redundant compilations.

For example, consider the use of `select:` to retrieve elements from a collection of Employees:

```
| fredEmps |
fredEmps := myEmployees select:
            [ :anEmployee | (anEmployee name) = 'Fred' ].
```

If `myEmployees` is a forwarder to a collection residing in the object server, then GemBuilder sends the parameter block's source code:

```
[ :anEmployee | (anEmployee name) = 'Fred' ].
```

to GemStone to be compiled and executed.

Replication of client Smalltalk blocks to GemStone Smalltalk is subject to certain limitations. When block replication violates one of these limitations, GemBuilder issues an error indicating that the attempted block replication has failed.

To avoid these limitations, consider using block callbacks instead. Block callbacks are discussed starting on page 113.

You can disable block replication completely using GemBuilder's configuration parameter **blockReplicationEnabled**. Block replication is enabled by default. Set this parameter to `false` to disable it, and GemBuilder raises an exception when block replication is attempted. This can be useful for determining if your application depends on block replication.

Image-stripping Limitations

Block replication relies on the client Smalltalk compiler and decompiler; if they've been removed from a deployed runtime environment, blocks cannot be replicated.

Two workarounds are possible:

1. Leave the compiler and decompiler in the image. For example, the VisualWorks Image Maker tool offers a "Remove Compiler" option which you can deselect to leave the compiler and decompiler in the image.
2. Do not use block replication. Usually this requires implementing a cover method for the block in a GemStone method, and sending that message instead. For instance:

   ```
   aForwarder select: [ :name | name = #Fred ]
   ```

—is instead coded:

```
aForwarder selectNameEquals: #Fred
```

...and in GemStone, `selectNameEquals:` is implemented as:

```
selectNameEquals: aName
  ^self select: [ :name | name = aName ]
```

When the block is encoded entirely in GemStone in this way, you can further optimize its operation by taking advantage of indexes and use an optimized selection block, as described in the *GemStone Programming Guide*.

Temporary Variable Reference Restrictions

A block is replicated in the form of its source code, without its surrounding context. Therefore, values drawn from outside the block's own scope cannot be relied upon to exist in both the client Smalltalk and in GemStone. Replication is not supported for blocks that reference instance variables, class variables, method arguments, or temporary variables declared external to the block's scope.

An exception is allowed in the case of global references, such as class names:

- Global variable references from inside a block must have the same name in both object spaces.

In the case of global variables containing data, it is the programmer's responsibility to ensure that the global identifier represents compatible values in both contexts.

Temporary variable reference restrictions disallow the following, because "tempName" is declared outside the block's scope:

```
| namedEmps tempName |
tempName := 'Fred'.
namedEmps := myEmployees select:
            [ :anEmployee | (anEmployee name) = tempName ].
```

As a workaround, implement a new Employees method in GemStone Smalltalk named `select:with:` that evaluates a two-argument block, in which the extra block argument is passed in as the `with:` parameter. For example:

```
select: aBlock with: extraArg
|result|

result := self speciesForSelect new.
self keysAndValuesDo: [ :aKey :aValue |
  (aBlock value: aValue value: extraArg) "two-value block"
    ifTrue: [result at: aKey put: aValue]
  ].

^ result.
```

You can then rewrite the application code to pass its temporary as the argument to the `with:` parameter without violating the scope of the block:

```
| namedEmps tempName |
tempName := 'Fred'.
namedEmps := myEmployees select:
            [ :anEmployee :extraArg |
                  (anEmployee name) = extraArg
            ] with: tempName.
```

Restriction on References to self or super

References to `self` and `super` are also context-sensitive and, therefore, disallowed:

- A replicated block cannot contain references to `self` or `super`.

For example, the following code cannot be forwarded to GemStone because the parameter block contains a reference to `self`:

```
myDict at:#key ifAbsent:[ self ]
```

References to `self` or `super` in forwarded code must occur outside the scope of the replicated block, where you can be sure of the context within which they occur. For example, you can rewrite the above code to return a result code, which can then be evaluated in the calling context, outside the scope of the replicated block:

```
result := myDict at:#key ifAbsent:[#absent].
result = #absent ifTrue: [ self ]
```

Explicit Return Restriction

Because a block is replicated without its surrounding context, a return statement has no surrounding context to which to return. Therefore:

- A replicated block cannot contain an explicit return.

For example:

```
result := myDict at:#key ifAbsent:[ ^nil ]
```

is disallowed. The statement can be recoded to perform its return within the calling context:

```
result := myDict at:#key ifAbsent:[#absent].
result = #absent ifTrue: [ ^nil ]
```

Replicating GemStone Blocks in Client Smalltalk

Also supported, though less commonly used, is the replication of GemStone blocks in client Smalltalk. Similar restrictions apply with regard to external references and the need for compiler / decompiler support. Blocks most frequently passed from the server to the client are the sort blocks that accompany instances of SortedCollection and its subclasses. Sort blocks rarely have occasion to violate replicated block restrictions.

If restrictions hamper you, consider using block callbacks instead.

Block Callbacks

Block callbacks provide an alternate mechanism for representing a client block in GemStone that avoids the limitations of block replication by calling back into the client Smalltalk to evaluate the block.

Block callbacks have the following advantages over block replication:

- Block callbacks don't require a compiler or decompiler.
- Block callbacks don't suffer the context limitations of block replication. The block can reference self, super, instance variables, and non-local temporaries; it can also perform explicit returns. For example, the following expression works correctly as a block callback, but fails if you try to replicate the block:

  aForwarder `at: aKey ifAbsent: [ ^nil ] asBlockCallback`

Block callbacks have the following disadvantages:

- A block that is evaluated many times in GemStone will perform poorly as a block callback. For example, the following expression sends a message to the client forwarder for each element of the collection represented by *aForwarder*:

  aForwarder `select: [ :e | e isNil ] asBlockCallback`

You can determine whether, by default, blocks are replicated or call back to the client using GemBuilder's configuration parameter **blockReplicationPolicy**. Legal values `#replicate` and `#callback`. A value of `#replicate` causes a client block to be stored in GemStone as a GemStone block. A value of `#callback` causes a client block to be stored in GemStone as a client forwarder, so that sending `value` to the block in GemStone causes `value` to be forwarded to the client block; the result of that block evaluation is then passed back to the GemStone context that invoked the block.

To ensure a specific replication policy for a given block, use the methods `asBlockCallback` or `asBlockReplicate`. Send `asBlockCallback` to ensure that the block always executes in the client, regardless of the default block

replication policy set by the configuration parameter. Likewise, send `asBlockReplicate` to ensure that the block is executed local to the context that invokes it (either in GemStone or in the client).For example:

```
dictionaryForwarder
   at: #X
   ifAbsent: [ ^nil ] asBlockCallback

collectionForwarder do: [ :e | e check ] asBlockReplicate
```

Replicating Collections with Instance Variables

If you create a subclass of a Collection and give it instance variables, you must reimplement the `copyEmpty:` method to ensure that added instance variables are included in the copy operation. Failure to reimplement `copyEmpty:` results in data loss.

For example, consider a Collection subclass called MyCollection that defines the additional instance variable `name`, with methods `name` and `name:` that retrieve and assign its value, respectively. MyCollection might reimplement `copyEmpty:` like this:

```
MyCollection >> copyEmpty: size

^(super copyEmpty: size) name: name
```

This reimplementation of `copyEmpty:` preserves the copying behavior of the superclass and assures that the added instance variable is also copied.

5.5 Precedence of Replication Mechanisms

Certain replication mechanisms can appear to contradict each other. The rules of precedence are:

- If the class methods `instVarMap` (for replicates) or `instancesAreForwarders` (for forwarders) are implemented, they take precedence over all others and are always respected.
- Otherwise, if the class method `replicationSpec` is implemented, or if an application calls or `replicationSpecSet:` to switch among several replication specs, those replication specs take precedence.

  In other words, if a class implements a replication spec, but it also implements `instancesAreForwarders` to return `true`, then instances of that class will be forwarders and the replication spec will be ignored.

Or, if a class implements both `instVarMap` and `replicationSpec`, the `instVarMap` determines which instance variables will be visible to the replication spec.

- In the absence of a replication spec, the instance method `faultToLevel:`, if called, is respected for replicates. Forwarders, of course, do not fault.
- For classes that use no other mechanism, the configuration parameters `faultLevelLnk` and `faultLevelRpc` are respected.

5.6 Evaluating Smalltalk Code on the GemStone server

In addition to sending messages to forwarders, GemBuilder provides mechanisms to execute ad-hoc Smalltalk code on the server.

Using the development environment Workspace, you can type in and select Smalltalk code and use the menu option “GS-Do it”, “GS-Inspect it” or “GS-Print it” to execute the selected text on the GemStone server, and return a replicate of the results.

You can also do this on the client by sending the string to a session for execution. The expression:

```
aGbsSession evaluate: aString
```

when executed on the client, tells GemBuilder to have the server compile and execute the GemStone Smalltalk code contained in *aString*, and answer a client replicate of the result of that execution. If, rather than a replicate, you would like the result as a forwarder, use the expression

```
aGbsSession fwevaluate: aString
```

The code in *aString* may be any arbitrary GemStone Smalltalk code that would compose a valid method body (see appendix A of the *GemStone Programming Guide* for GemStone Smalltalk syntax), with the exceptions that the code:

- ❒ cannot take any arguments
- ❒ must not refer to the variables self or super
- ❒ must not refer to any instance variable of any class

Example 5.9 shows how to use evaluate: to execute code.

Example 5.9

```
resultReplicate := GBSM currentSession
   evaluate: '
      | result |
      result := Array new: 3.
      result
         at: 1 put: ''Pear'';
         at: 2 put: #unripe;
         at: 3 put: 42.
      ^ result'
```

You can avoid some of these restrictions by passing in a context object using:

`aGbsSession evaluate:` *aString* `context:` *aServerObject*

or

`aGbsSession fwevaluate:` *aString* `context:` *aServerObject*

The context argument, *aServerObject*, can be any replicate of or forwarder to a GemStone server object. If the code in *aString* refers to the variables self or super, these will be bound to the context object. The code in *aString* can also refer to any instance variables of the context object.

Example 5.10

```
aGbsSession
   evaluate: 'self at: 2 put: #ripe'
   context: resultReplicate.
```

The advantage of the `evaluate:` family of messages is that they allow you to execute arbitrary ad-hoc code on the server without previously defining a method.

However, this isn't always the best way to execute server code. The `evaluate:` messages invoke the GemStone Smalltalk compiler upon each execution, and so have extra overhead. Also, the inability to pass arguments rules out the evaluate: messages for some uses.

Message sends through forwarders are the most common means of initiating execution of GemStone Smalltalk code on the server. However, a message passed through a forwarder will fail if the server object that receives the message does not understand that message. Forwarder sends require previous definition of an appropriate GemStone method on the server.

The two forms of execution complement each other. The `evaluate:` messages do not require prior method definition, but cannot take arguments. Forwarder sends require prior method definition, but can take arguments.

5.7 Converting Between Forms

A variety of messages exist to convert between delegates, forwarders, replicates, stubs, and unconnected client objects. Table 5.1–Table 5.5 list the results of sending any of several conversion messages to these objects.

A delegate is an instance of GbxDelegate. Delegates are used internally by GemBuilder. An application doesn't normally need to use delegates directly, but you may see them when debugging. We recommend against using delegate protocol, as in Table 5.1, in customer applications.

NOTE

To avoid unpredictable consequences and possible errors, do not use the expressions listed as producing undefined results.

Table 5.1 Delegate Conversion Protocol

Message	Return Value
`copy`	shallow copy of delegate
`asLocalObject`	replicate
`asGSObject`	`self`
`asForwarder`	undefined
`beReplicate`	undefined
`fault`	undefined
`stubYourself`	undefined

Table 5.2 Forwarder (to the Server) Conversion Protocol

Message	Return Value
`copy`	copies associated server object and returns replicate of copy
`asLocalObject`	undefined
`asGSObject` (not recommended for customer applications)	associated delegate
`asForwarder`	`self`
`beReplicate`	`self`, which has become a replicate
`fault`	`self` (use `beReplicate` to make a replicate)
`stubYourself`	`self`

Table 5.3 Replicate Conversion Protocol

Message	Return Value
`copy`	shallow copy of delegate not associated with any server object
`asLocalObject`	undefined
`asGSObject` (not recommended for customer applications)	associated delegate
`asForwarder`	`self`, which has become a forwarder
`beReplicate`	`self`
`fault`	`self`, whose instance variables are now also replicates to the configured fault level
`stubYourself`	`self`, which has become a stub

Table 5.4 Stub Conversion Protocol

Message	Return Value
`copy`	shallow copy; receiver becomes a replicate
`asLocalObject`	undefined
`asGSObject` (not recommended for customer applications)	associated delegate
`asForwarder`	`self`, which has become a forwarder
`beReplicate`	`self` (use `fault` to become a replicate)
`fault`	`self`
`stubYourself`	`self`

Table 5.5 Conversion Protocol for Unshared Client Objects

Message	Return Value
`copy`	shallow copy
`asLocalObject`	undefined
`asGSObject` (not recommended for customer applications)	new delegate; creates new associated server object
`asForwarder`	`self`, which has become a forwarder; creates new associated server object
`beReplicate`	`self`
`fault`	`self`
`stubYourself`	`self`

Chapter

6 Connectors

This chapter describes *connectors*, which allow an application developer to explicitly declare an association between a root client object and a root server object.

- Connectors function at login. After that, you must manually disconnect and reconnect them to effect any changes.
- There are different kinds of connectors for different types of objects.
- Connectors may update either connected object, depending on how they are set up.
- Connectors exist either in a given session, or globally—in every session your image defines.

Connecting Root Objects
explains which objects to associate using connectors.

Connecting and Disconnecting
describes what connectors do and when they do it.

Kinds of Connectors
describes the available kinds of connectors and the differences between them.

Making and Managing Connectors
explains how to use connectors in code or using the Connector Browser.

6.1 Connecting Root Objects

Every replicate and forwarder in the client is connected to an object in the server. You do not, however, need a connector for every replicate or forwarder. A typical application only needs connectors for a small number of root objects. A connector connects more than the specified client object to the specified server object: through transitive reference, a connector connects whole networks of objects. Most objects (except atomic objects—characters, booleans, small integers, `nil`) refer to others through their instance variables. And their instance variables refer to *their* instance variables, and so on, branch and twig, until you reach the leaves of a large network of objects with a treelike structure.

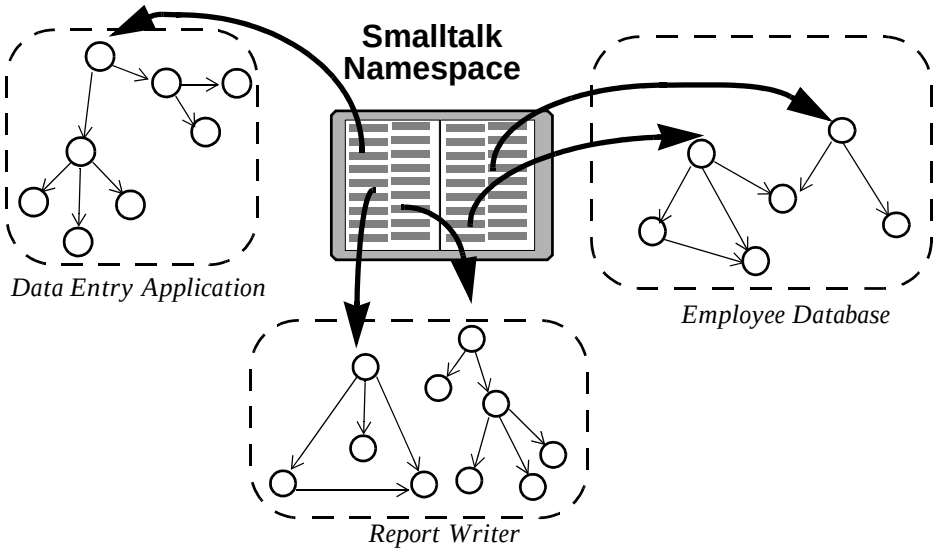
You can take advantage of this hierarchical structure to minimize application overhead. Identify the object at the root of each subsystem of shared objects, and then connect only these root objects. Depending on how you've defined configuration parameters and related matters, you can synchronize entire subsystems in GemStone/S this way. After you've connected the application's roots, GemBuilder automatically manages all the objects referenced from these roots.

Root objects are often:

- global variables,
- class or shared variables, or
- class instance variables.

Figure 6.1 shows an application in which several connected objects are accessed through global or shared variables in client Smalltalk. One system represents an employee database. Another system represents a data entry application for creating and modifying objects. A third system represents a report writer for these objects. Dotted lines in the figure group the logically related subsystems.

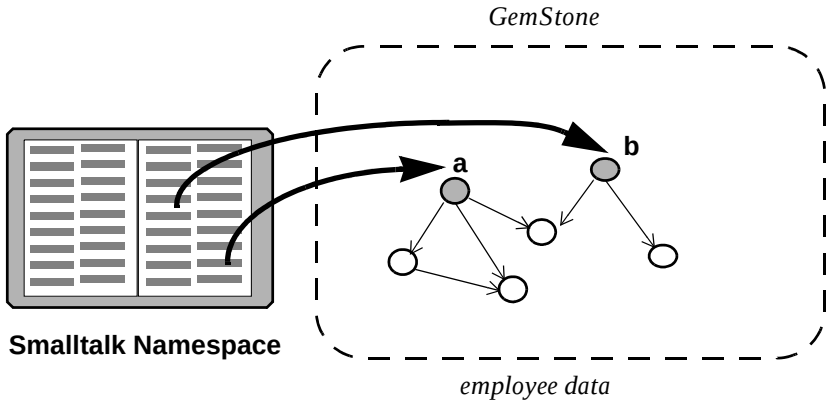
Figure 6.1 Connecting Application Roots



The data entry application and the report writer reside in the client Smalltalk image; however, the employee database is stored on the GemStone server, as it defines a large amount of persistent data that other users may need to share, data that benefits from GemStone/S's capacity, stability, robustness, and fast searches.

Figure 6.2 shows the state of the employee data when stored on the server:

Figure 6.2 Root Objects



In Figure 6.2, objects **a** and **b** are *root* objects: those objects from which all others can be reached by *transitive closure*: by direct reference, or by indirect reference through any number of layers.

The above discussion has focused on shared instances from your applications, but in order to share instances in any way, GemBuilder and GemStone must first share definitions for each class of shared instance.

Scope

Some connectors connect their objects whenever any session logs in; some do so only when a specific session logs in:

- *Global connectors* allow you to maintain a standard set of connectors common to all applications in your GemBuilder image.
- *Session connectors* allow individual applications to customize connectors: you define unique session parameters for each application, and different sessions can connect different objects. When sessions of one kind log in, other sessions' connectors are defined but not connected.

When a session logs in, its session connectors and all global connectors (if not already connected) connect automatically.

When a session logs out, its session connectors disconnect. If the session is the last in the application to log out, it disconnects the global connectors.

Verifying Connections

Connectors are saved in client Smalltalk sets, separate ones for global connectors and each named kind of session (each uniquely defined set of session parameters). Two connectors are considered equal if they resolve to the same client object. Client Smalltalk sets eliminate duplicates based on equality. Therefore:

NOTE

Adding a global or session connector that points to the same object as an existing connector will remove the existing connector.

Duplicate session connectors are not removed if they are stored in different sessions.

GemBuilder provides a configuration parameter, **`connectVerification`**, that, when `true`, causes connectors to verify at login that they are not redefining a connector that already exists. In addition, class connectors verify that the two classes they are connecting have compatible structures.

If a connector fails verification, GemBuilder issues a notifier if **verbose** is also `true`, or raises an exception otherwise. You can set **`connectVerification`** in the Connector Browser or in the Settings Browser.

NOTE

Under certain circumstances, GemBuilder may verify that a connector still exists in a session after it has been explicitly removed from that session. This is because the order in which connectors are connected at login is not deterministic.For more information and a work-around, see *"Connection Order" on page 128.*

Initializing

At login, a connector associates an object in a single-user image with an object in a multiuser repository. The value of either could have changed since last login. Which value is valid?

Connectors can initialize either object by performing a specified *postconnect action:*

Update Smalltalk
default for all but class connectors, initializes the client object with the current state of the GemStone server object.

Update GemStone
initializes the GemStone server object with the current state of the client object.

Forward to the server or client
makes one object a forwarder to the other. Forwarders are discussed starting on page 86.

No initialization
leaves the client object and GemStone server object unmodified after connection—default for class connectors.

As the name implies, postconnect actions execute only at initial connection. After that, changes propagate according to mark dirty specifications, as described in "Synchronizing State" on page 90, or they do not propagate at all, as is normally the case with class connectors, as described in "Class Mapping" on page 83.

Updating Class Definitions

By default, after login and initialization, class connectors do not propagate changes. If you've defined classes differently on the client and the server, you probably had good reason to do so; you probably don't want one object space to update the other with its own class definition. Therefore, to avoid updating class definitions, class connectors generally specify a postconnect action of *none.*

For similar reasons, class connectors should not, and in fact cannot, specify that the client class is a forwarder—the forwarder and clientForwarder postconnect action are unavailable for class connectors.

If you change either a client or GemStone class definition during a session, you must propagate the change yourself by disconnecting and reconnecting the connector. The Connector Browser, described starting on page 136, provides convenient buttons for the purpose.

NOTE

Remember to restore a postconnect action of none *after you complete the desired update.*

6.2 Connecting and Disconnecting

At login, connectors connect objects according to their specifications; thereafter, they are inactive. Changes to instances that occur during the course of a session are replicated either because those instances are synchronized replicates that mark changes dirty, or because one is a forwarder to the other. Changes to class definitions or other unsynchronized changes must be propagated manually. To do so, use the **Disconnect** and **Connect** buttons in the Connector Browser to disconnect and reconnect the appropriate connector.

At logout, GemBuilder sets the instance variables of connectors to nil, if the GemBuilder configuration parameter **connectorNilling**,is set to true (the default). This reduces the risk of defunct stub or forwarder errors, replacing them with `nil doesNotUnderstand` errors.

Only connectors whose values are set from the server on login are cleared when **connectorNilling** is true. Session-based name, class variable, or class instance variable connectors that have a postconnect action of `#updateST` or `#forwarder` are cleared. Fast connectors, class connectors, or connectors whose postconnect action is **#updateGS** or **#none** do not have instanace variables set to nil.

connectorNilling can be set for individual sessions, if desired. See “Setting Configuration Parameters” on page 215 for details on setting session-specific configuration parameters. The detailed description for this configuration parameter is on page 222.

6.3 Kinds of Connectors

Five kinds of connectors use different ways of finding the two objects to connect. You have already encountered one kind:

Class connector — connects a client Smalltalk and GemStone class. As discussed in “Class Mapping” on page 83, to replicate an object, both client and repository must define the class, and the two classes must be connected using a class connector.

For replicating instances, however, we need ways to connect root objects:

Name connector — connects client and server objects identified by name. Figure 6.3 illustrates how a name connector connects a client object to a server object.

Class variable connector — first resolves the named objects representing the classes, then looks for a class variable in the GemStone class, and a Class or Shared Variable in the client Smalltalk class with the specified name and connects those objects.

Class instance variable connector — first resolves the named objects representing the classes, then looks for a class instance variable in each class with the specified name and connects those objects.

Fast connector — connects the GemStone kernel classes to their client Smalltalk counterparts. Fast connectors are predefined, implemented for speed. The kernel classes to which they point will not change

identity during the course of a session; GemBuilder can take advantage of that to reduce overhead. The GemStone kernel class connectors are predefined, and GemBuilder relies on them.

NOTE

Application objects can change identity during the course of a session. Applications should therefore not define fast connectors.

Connection Order

At login, GemBuilder connects connectors in the following order:

1. First, predefined fast connectors for kernel classes;
2. next, class connectors whose postconnect action is anything other than **updateGS**; and finally
3. all other connectors, in no particular order.

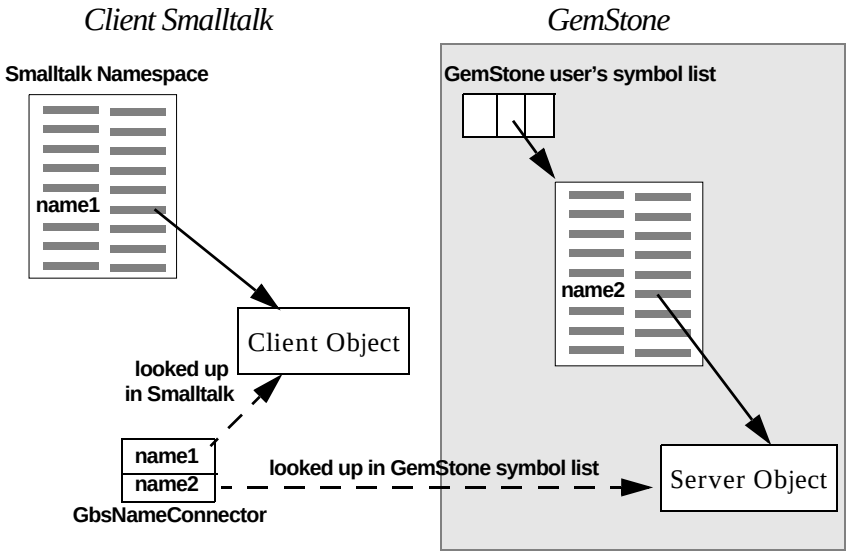
You can control the order in which connectors connect by connecting them explicitly in your code, instead of relying on GemBuilder's automatic mechanism to connect them for you at login.

Lookup

Names must be found in namespaces. In the client, objects included in a Smalltalk namespace can be visible throughout the Smalltalk image. GemStone implements namespaces with symbol dictionaries: if the symbol list of the session user includes the symbol dictionary defining object A, then object A is visible to that user.

Lookup occurs when the connection is established—that is, when the session first logs in.

Figure 6.3 Connecting a Name Connector



Connecting by Identity: Fast Connectors

Name lookup in both client Smalltalk and GemStone Smalltalk can be slow if you are using a lot of connectors. You can bypass the name lookup by using a fast connector, which saves direct references to the client Smalltalk objects and the object IDs of the GemStone server objects that are connected.

Using fast connectors can be risky, however. If the GemStone server object is renamed or redefined, a fast connector will continue to point to the old object: the one with the same object identifier. When the identity of an object changes (for example, if it is a variable that you assign to a new object), a fast connector becomes incorrect. An out-of-date fast connector may cause an "object does not exist" error, or it may silently continue to pass messages to an old object.

Because using object identity is not always an appropriate way to resolve an object, we recommend that you generally use standard connectors instead of fast connectors, especially during early development stages. You can always use the Connector Browser to change a connector type later, when you are certain that your application can rely on named objects to have a constant identity.

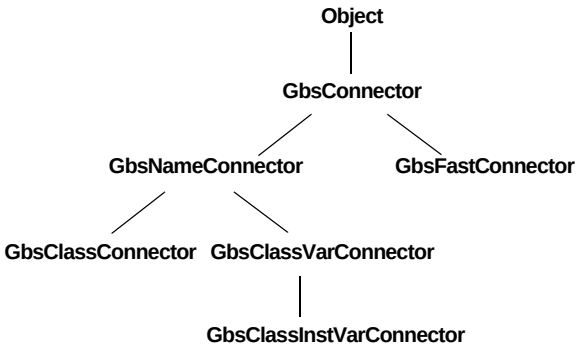
6.4 Making and Managing Connectors

To make and manage connectors interactively, see “The Connector Browser” on page 136. The next section describes making and managing connectors in code.

Making Connectors Programmatically

GbsConnector is the abstract superclass for the connector class hierarchy. These classes implement connection methods and define instance variables to refer to the associated GemStone and client objects. Figure 6.4 shows the hierarchy.

Figure 6.4 Connector Class Hierarchy



To create a connector programmatically:

1. Create the connector.
2. Set its postconnect action, if other than the default.
3. Add it to the global connector list, or a connector list for session parameters.

Create the required GemStone session parameters and connectors in an initialization method.(Creation methods for session parameters are described in “Session Parameters” on page 35.)

Creating Connectors

One simple creation method for a name connector requires only the names of the two objects to be connected:

```
GbsNameConnector stName: stName
                 gsName: gsName
```

You can create a class connector this way too:

```
GbsClassConnector stName: stName
                  gsName: gsName
```

The above methods require that the server object already exist. If GemBuilder must create the object, choose an instance creation method that specifies the GemStone server dictionary in which to place it:

```
GbsNameConnector stName: stName
   gsName: gsName
   dictionaryName: gsDictionary
```

To create a class variable connectors:

```
GbsClassVarConnector
   stName: #ClassName
   gsName: #ClassName
   cvarName: #ClassVarName
```

Similarly, a class instance variable connector:

```
GbsClassInstVarConnector
   stName: #ClassName
   gsName: #ClassName
   cvarName: #ClassInstVarName
```

For more, browse instance creation methods for each connector class.

Setting the Postconnect Action

The symbolic names for postconnect actions are `#updateST`, `#updateGS`, `#forwarder`, `#clientForwarder`, and `#none`. All connectors default to using `#updateST` except class connectors, which default to `#none`.

To cause a GemStone server object to take its initial values at login from its Smalltalk counterpart, send `postConnectAction: #updateGS` to the connector. This is occasionally useful for loading data into GemStone from the client image.

Adding Connectors to a Connector List

When you create a connector, you must decide whether it is to be managed by an individual session or globally. Leaving it unmanaged can have several adverse effects: it will not be connected and disconnected when required, and object retrieval may slow.

A connector is managed by adding it to the appropriate list of connectors.

If you want a connector in effect whenever any session logs in, put it in the global connectors collection:

```
GBSM addGlobalConnector: aConnector
```

A new global connector first takes effect the next time any session logs in.

Each session parameters object maintains its own list of session connectors. If you want a connector in effect whenever a session logs in using specific parameters, add a connector to the session parameters object:

```
ThisApplicationParameters addConnector: aConnector
```

A new session connector first takes effect the next time that session logs in.

To initialize a system with two roots, the global `BigDictionary`, and a class variable in `MyClass` called `MyClassVar`, your application might execute code such as that shown in Example 6.1:

Example 6.1

```
GBSM  addGlobalConnector: (GbsNameConnector
            stName: #MyGlobal
            gsName: #MyGlobal);
      addGlobalConnector: (GbsClassVarConnector
            stName: #MyClass
            gsName: #MyClass
            cvarName: #MyClassVar)
```

Initialization code such as that in Example 6.1 needs to execute only once. From then on, every time you log into GemStone, `MyGlobal` and `MyClassVar` (and all the objects they reference) connect; after that, replication and updating occur as specified.

Session Control

The following examples illustrate one approach to managing GemBuilder sessions and connectors: a session control class that defines these methods for, in this example, a help request system.

An instance of the session control class could be stored in the application object as a class variable, in which case the session information would be the same for all instances of the application, or it could be stored in the application as an instance variable, in which case each instance of the application would get its own copy to change as needed. In either case, methods to create the session parameters object and its connectors might follow these patterns:

Example 6.2 shows the method `session`, which returns the application's logged-in session. If the session is not logged in, the method requests an RPC login and returns the resulting session. If login fails, the method returns `nil`.

Example 6.2

```
session
      "self session"
      (session isNil or: [session isLoggedIn not]) ifTrue: [
            session := self sessionParameters loginRpc.
            session isNil ifTrue: [^nil]].
      ^session
```

Example 6.3 shows a method that initializes a set of session parameters. (For security, you may choose to prompt for passwords instead.)

Example 6.3

```
sessionParameters
      | params |
      sessionParameters isNil ifTrue: [
            params := GbsSessionParameters new.
            params gemStoneName: 'gs64stone'.
            params username: 'DataCurator'.
            params password: 'swordfish'.
            params gemService: 'gemnetobject'.
            params rememberPassword: true.
            params rememberHostPassword: true.
            self addConnectorsTo: params.
            sessionParameters := params.
            GBSM addParameters: params].
      ^sessionParameters
```

Example 6.4 adds connectors to the session parameters object by calling lower-level methods to individual types of connectors:

Example 6.4

```
addConnectorsTo: aParams
      self addClassConnectorsTo: aParams.
      self addClassVarConnectorsTo: aParams
```

Example 6.5 shows a method that creates class connectors and adds them to the session parameters connector list:

Example 6.5

```
addClassConnectorsTo: aParams
      aParams addConnector:
            (GbsClassConnector
                  stName: #GST_Action
                  gsName: #GST_Action).
      aParams addConnector:
            (GbsClassConnector
                  stName: #GST_Customer
                  gsName: #GST_Customer).
      aParams addConnector:
            (GbsClassConnector
                  stName: #GST_Engineer
                  gsName: #GST_Engineer).
```

Example 6.6 shows a method that creates class variable connectors and adds them to the session parameters connector list:

Example 6.6

```
addClassVarConnectorsTo: aParams
      | aConnector |
      aParams addConnector:
            (aConnector := GbsClassVarConnector
                  stName: #GST_HelpRequest
                  gsName: #GST_HelpRequest
                  cvarName: #AllRequests).
            aConnector postConnectAction: #forwarder.
      aParams addConnector:
            (GbsClassVarConnector
                  stName: #GST_Company
                  gsName: #GST_Company
                  cvarName: #AllCompanies)
```

You can create methods similar to those shown in examples 6.5 and 6.6 to create name connectors and global connectors for your application, as well.

NOTE

If more than one session is logged into GemStone using the same session parameters object, and you add a connector to one of those sessions,

GemBuilder will try to connect that connector for all sessions sharing the same parameters. If any fail to reference the GemStone server object represented by the connector, you'll receive an error message stating that the connector failed to connect.

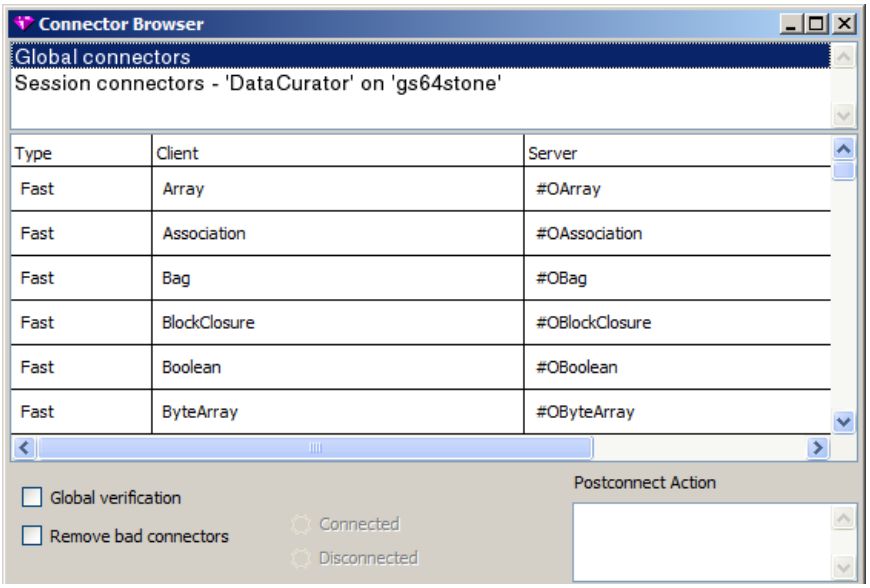
The Connector Browser

You can use GemBuilder's Connector Browser to make and manage connectors interactively. To open a Connector Browser, select **Connectors** from the GemStone menu. With this browser, you can:

- examine, create, and remove global or session-based connectors;
- inspect the client or server object to which a connector resolves;
- determine whether a specified connection is currently connected;
- connect or disconnect a connector; and
- examine or modify the postconnect action associated with a connector.

Figure 6.5 shows the Connector Browser.

Figure 6.5 The Connector Browser



The Group Pane

The top pane is the Group pane; it allows you to select either global connectors or those associated with an individual session. Global connectors are predefined to connect the GemStone server kernel classes with their client Smalltalk counterparts. When you select an item in this pane, the connectors defined for the selected item appear in the middle pane.

In the Group pane, the popup menu provides the following items:

Table 6.1 Group List Menu in the Connector Browser

update	Refreshes the views and updates the browser; useful if you have made changes in other windows and need to synchronize the browser with them.
initialize	*(available only when Global Connectors are selected)* Allows you to remove all connectors except those that connect kernel classes.

The Connector Pane

The middle pane is the Connector pane; it lists the connectors, their types, and descriptions in both the client and GemStone server Smalltalks. In the Connector pane, the popup menu offers the following items:

Table 6.2 Connectors Menu in the Connector Browser

inspect Client	Resolves and inspects the client Smalltalk object for the selected connector.
inspect Server	Resolves and inspects the GemStone server object for the selected connector.
add...	Adds a new connector, prompting for required information.
remove...	Removes a connector, after confirmation.

The Control Panel

The bottom pane is a control panel that allows you to change the **connectVerification** and **removeInvalidConnectors** configuration parameters and connect or disconnect objects. setting a connector's postconnect action is described in the section that follows.

Table 6.3 Options in the Control Panel

Global verification	When enabled, connectors (other than class connectors) verify that they are not redefining an object connection before connecting. Class connectors, upon connection, verify that the structures of the two connected classes are of the same storage type.
Remove bad connectors	When enabled, connectors that fail to resolve at login are automatically removed from the connector collections.
Connected / Disconnected	Connects or disconnects the GemStone and client Smalltalk objects described by the connector. Applies to the selected session, or to the current session if global connectors are selected.

Enabling connector verification can slow login: we recommend that you turn on verification during development and turn it off for production systems.

Postconnect Action

The postconnect action determines how GemBuilder sets the initial state of connected objects. Options are:

Table 6.4 Postconnect Action Options in the Connector Browser

updateST	Initializes the client object using the current state of the GemStone server object.
updateGS	Initializes the GemStone server object using the current state of the client object.
forwarder	Makes the client object a forwarder to the GemStone server object.
clientForwarder	Makes the GemStone server object a forwarder to the client object.
none	Leaves the client object and the GemStone server object unchanged after their initial connection. The connector in the case has no function, so it's unlikely you'll want to use this option.

To create a new connector:

1. Select the session in the Group pane.
2. Place the cursor in the Connector pane.
3. Select **add** from the menu.
4. When prompted, specify the type of connector.
5. When prompted, specify the names of the client and server objects.
6. When prompted, specify the name of the dictionary for the server object.
7. Specify the postconnect action.

To create a forwarder:

1. Create a connector as described above.
2. Select **forwarder** as the desired postconnect action.

To change the postconnect action:

1. Disconnect the objects by clicking on the **Disconnected** button.
2. Change the postconnect action as required.
3. Reconnect the objects by clicking on the **Connected** button.

If your application initially stores its data in the client, and you intend to store the data on the GemStone server but have not done so yet:

1. Create a connector or connectors for the root object(s) in the data set.
2. Select **updateGS** as the postconnect action for these connectors.
3. Log into the GemStone server so that GemBuilder can create the GemStone server replicates for the client Smalltalk data.
4. Inspect the GemStone server objects to be sure they have the intended values.
5. Commit the transaction and log out.
6. Select the connectors and change their postconnect actions to **updateST** so that future sessions will begin by using the stored GemStone server data.

Chapter 7

Managing Transactions

The GemStone object server's fundamental mechanism for maintaining the integrity of shared objects in a multiuser environment is the *transaction*. This chapter describes transactions and how to use them. For further information, see the chapter in the *GemStone Programming Guide* entitled "Transactions and Concurrency Control."

Transaction Management: an Overview
introduces the concepts to be explained later in the chapter.

Operating Inside a Transaction
explains the transaction model, committing, and aborting.

Operating Outside a Transaction
discusses a lower-overhead alternative for read-only views of the shared repository.

Transaction Modes
explains the difference between automatic and manual transaction modes.

Managing Concurrent Transactions
discusses concurrency conflicts and ways to minimize them, such as locks.

Reduced-conflict Classes
describes specialized GemStone collections that minimize conflicts without locking.

Changed Object Notification
explains a mechanism for coordinating the activities of multiple sessions.

7.1 Transaction Management: an Overview

The GemStone object server provides an environment in which many users can share the same persistent objects. The object server maintains a central repository of shared objects. When a GemBuilder application needs to view or modify shared objects, it logs in to the GemStone object server, starting a session as described in Chapter 2.

A GemBuilder session creates a private view of the GemStone repository containing views of shared objects for the application's use. The application can perform computations, retrieve objects, and modify objects, as though it were a single-user Smalltalk image working with private objects. When appropriate, the application propagates its changes to the shared repository so those changes become visible to other users.

In order to maintain consistency in the repository, GemBuilder encapsulates a session's operations (computations, fetches, and modifications) in units called *transactions.* Any work done while operating in a transaction can be submitted to the object server for incorporation into the shared object repository. This is called *committing* the transaction.

During the course of a logged-in session an application can submit many transactions to the GemStone object server. In a multiuser environment, concurrency conflicts will arise that can cause some commit attempts to fail. *Aborting* the transaction refreshes the session's view of the repository in preparation for further work.

In order to reduce its operating overhead, a session can run *outside a transaction*, but to do so the session must temporarily relinquish its ability to commit. A session running outside a transaction operates in *manual transaction mode*, in contrast to the system default *automatic transaction mode*.

Another potential mode is *transactionless mode.* However, this mode is not usable from within GemBuilder.

GemBuilder provides ways of avoiding the concurrency conflicts that can cause a commit to fail. *Optimistic concurrency control* risks higher rates of commit failure in

exchange for reduced transaction overhead, while *pessimistic concurrency control* uses locks of various kinds to improve a transaction's chances of successfully committing. GemBuilder also offers *reduced-conflict classes* that are similar to familiar Smalltalk collections, but are especially designed for the demands of multiuser applications.

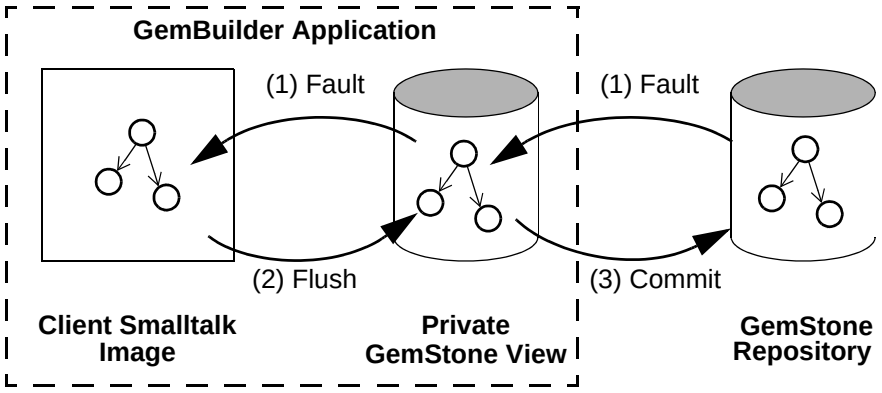
This chapter explains each of the topics mentioned here: transactions, committing and aborting, running outside a transaction, automatic and manual transaction modes, optimistic and pessimistic concurrency control, and reduced conflict classes. Be sure to refer to the related topics in the *GemStone Programming Guide* for a full understanding of these transaction management concepts.

7.2 Operating Inside a Transaction

While a session is logged in to the GemStone object server, GemBuilder maintains a private view of the shared object repository for that session. To prevent conflicts that can arise from operations occurring simultaneously in different sessions in the multiuser environment, GemBuilder encapsulates each session's operations in a *transaction*. Only when the session commits its transaction does GemStone try to merge the modified objects in that session's view with the main, shared repository.

Figure 7.1 shows a client image and its repository, along with a common sequence of operations: (1) faulting in an object from the shared repository to Smalltalk, (2) flushing an object to the private GemStone view, and (3) committing the object's changes to the shared repository.

Figure 7.1 GemBuilder Application Workspace



The private GemStone view starts each transaction as a snapshot of the current state of the repository. As the application creates and modifies shared objects, GemBuilder updates the private GemStone view to reflect the application's changes. When your application commits a transaction, the repository is updated with the changes held in your application's private GemStone view.

For efficiency, GemBuilder does not replicate the entire contents of the repository. It contains only those objects that have been replicated from the repository or created by your application for sharing with the object server. Objects are replicated only when modified. This minimizes the amount of data that moves across the boundary from the repository to the Smalltalk application.

Committing a Transaction

When an application submits a transaction to the object server for inclusion in the shared repository, it is said to *commit* the transaction. To commit a transaction from the client, send the message:

aGbsSession `commitTransaction` (to commit a specific session)

or:

`GBSM commitTransaction` (to commit the current session)

or, in the Session Browser, select the session and click on the **Commit...** button; or in the Class Browser, use the pop up menu on the SymbolDictionary pane to select **commit**.

When the commit succeeds, the method returns `true`. Successfully committing a transaction has two effects:

- It copies the application's new and changed objects to the shared object repository, where they are visible to other users.
- It refreshes the application's private GemStone view by making visible any new or modified objects that have been committed by other users.

A commit request can be unsuccessful in two ways:

- A commit *fails* if the object server detects a concurrency conflict with the work of other users. When the commit fails the `commitTransaction` method returns `false`.
- A commit is *not attempted* if a related application component is not ready to commit. When the commit is not attempted, the `commitTransaction` method returns `nil`. (See "Session Dependents" on page 44.)

In order to commit, the session must be operating within a transaction. An attempt to commit while outside a transaction raises an exception.

Aborting a Transaction

A session refreshes its view of the shared object repository by aborting its transaction. Despite the terminology, a session need not be operating inside a transaction in order to abort. To abort, send the message:

aGbsSession `abortTransaction` (to abort a specific session)

or:

`GBSM abortTransaction` (to abort the current session)

or, in the Session Browser, select a logged-in session and click on the **Abort...** button; or in the Class Browser, use the pop up menu on the SymbolDictionary pane to select **abort**.

Aborting has these effects:

- The transaction (if any) ends. If the session's transaction mode is automatic, GemBuilder starts a new transaction. If the session's transaction mode is manual, the session is left outside of a transaction.
- Temporary Smalltalk objects remain unchanged.
- The session's private view of the GemStone shared object repository is updated to match the current state of the repository.

Avoiding or Handling Commit Failures

You can use the GemBuilder method GbsSession >> `hasConflicts` to determine if any concurrency conflicts exist that would cause a subsequent commit operation to fail. It returns `false` if it finds no conflicts with other concurrent transactions, `true` otherwise. You can then determine how best to proceed.

If an attempt to commit fails because of a concurrency conflict, the `commitTransaction` method returns `false`.

Following a commit failure, the client's view of persistent objects may differ from their precommit state:

- The current transaction is still in effect. However, you must end the transaction and start a new one before you can successfully commit.
- Temporary Smalltalk objects remain unchanged.

- Modified GemStone server objects remain unchanged.
- Unmodified GemStone server objects are updated with new values from the shared repository.

Following a commit failure, your session must refresh its view of the repository by aborting the current transaction. The uncommitted transaction remains in effect so you can save some of its contents, if necessary, before aborting.

A common strategy for handling such a failure is to abort, then reinvoke the method in which the commit occurred. Depending on your application, you may simply choose to discard the transaction and move on, or you may choose to remedy the specific transaction conflict that caused the failure, then initiate a new transaction and commit.

If you want to know why a transaction failed to commit, you can send the message:

aGbsSession `transactionConflicts`

This expression returns a symbol dictionary whose keys indicate the kind of conflict detected and whose values identify the objects that incurred each kind of conflict. (See “Managing Concurrent Transactions” on page 150 for more discussion of the kinds of conflicts that can arise.)

7.3 Operating Outside a Transaction

A session must be *inside a transaction* in order to commit. While operating within a transaction, every change the session makes and every new object it creates can be a candidate for propagation to the shared repository. GemBuilder monitors the operations that occur within the transaction, gathering all the necessary information required to prepare the transaction to be committed.

For efficiency, an application may configure a session to operate *outside a transaction*. When operating outside a transaction, a session can view the repository, browse the objects it contains, and even make computations based upon their values, but it cannot commit any new or changed GemStone server objects. While operating outside a transaction, a session saves some of the overhead of tracking changes, which may be significant in some applications. A session operating outside a transaction can, at any time, begin a transaction.

No session is overhead-free: even a session operating outside a transaction uses GemStone resources to manage its objects and its view of the repository. For best system performance, all sessions, even those running outside a transaction, must periodically refresh their views of the repository by committing or aborting.

Table 7.1 shows GbsSession methods that support running outside of a GemStone transaction:

Table 7.1 GbsSession Methods for Running Outside of a Transaction

`beginTransaction`	Aborts and begins a transaction.
`transactionMode`	Returns `#autoBegin` or `#manualBegin`
`transactionMode:`*newMode*	Sets `#autoBegin` or `#manualBegin`
`inTransaction`	Returns `true` if the session is currently in a transaction.
`signaledAbortAction:` *aBlock*	Executes *aBlock* when a signal to abort is received (see below).

To begin a transaction, send the message:

aGbsSession `beginTransaction`
(to begin a transaction for a specific session)

or:

`GBSM beginTransaction`
(to begin a transaction for the current session)

or, in the Session Browser, select a logged-in session and click on the **Begin...** button.

This message gives you a fresh view of the repository and starts a transaction. When you abort or successfully commit this new transaction, you will again be outside of a transaction until you either explicitly begin a new one or change transaction modes.

If you are not currently in a transaction, but still want a fresh view of the repository, you can send the message *aGbsSession* `abortTransaction`. This aborts your current view of the repository and gives you a fresh view, but does not start a new transaction.

Being Signaled to Abort

When you are in a transaction, GemStone waits until you commit or abort to reclaim storage for objects that have been made obsolete by your changes. When you are running outside of a transaction, however, you are implicitly giving GemStone permission to send your Gem session a signal requesting that you abort your current view so that GemStone can reclaim storage when necessary. When this happens, you must respond within the time period specified in the

STN_GEM_ABORT_TIMEOUT parameter in your configuration file. If you do not, GemStone either terminates the Gem or forces an abort, depending on the value of the related configuration parameter STN_GEM_LOSTOT_TIMEOUT. The Stone forces an abort by sending your session an `abortErrLostOtRoot` signal, which means that your view of the repository was lost, and any objects that your application had been holding may no longer be valid. When your application receives `abortErrLostOtRoot`, the application must log out of GemStone and log back in, thus rereading all of its data in order to resynchronize its snapshot of the current state of the GemStone repository.

You can avoid `abortErrLostOtRoot` and control what happens when you receive a signal to abort with the `signaledAbortAction`: *aBlock* message. For example:

```
aGbsSession signaledAbortAction:
    [aGbsSession abortTransaction].
```

This causes your GemBuilder session to abort when it receives a signal to abort.

7.4 Transaction Modes

A GemBuilder session always initiates a transaction when it logs in. After login, the session can operate in either of two transaction modes: automatic or manual.

Automatic Transaction Mode

In *automatic transaction mode*, committing or aborting a transaction automatically starts a new transaction. This is GemBuilder's default transaction mode: in this mode, the session operates within a transaction the entire time it is logged into GemStone.

However, being in a transaction incurs certain costs related to maintaining a consistent view of the repository at all times for all sessions. Objects that the repository contained when you started the transaction are preserved in your view, even if you are not using them and other users' actions have rendered them meaningless or obsolete.

Depending upon the characteristics of your particular installation (such as the number of users, the frequency of transactions, and the extent of object sharing), this burden can be trivial or significant. If it is significant at your site, you may want to reduce overhead by using sessions that run outside transactions. To run outside a transaction, a session must switch to manual transaction mode.

Manual Transaction Mode

In *manual transaction mode*, the session remains outside a transaction until you begin a transaction. When you change the transaction mode from automatic (its initial setting) to manual, the current transaction is aborted and the session is left outside a transaction. In manual transaction mode, a transaction begins only as a result of an explicit request. When you abort or commit successfully, the session remains outside a transaction until a new transaction is initiated.

To begin a transaction, send the message

aGbsSession `beginTransaction`

or select the **Begin...** button on the Session Browser.

A new transaction always begins with an abort to refresh the session's private view of the repository. Local objects that customarily survive an abort operation, such as temporary results you have computed while outside a transaction, can be carried into the new transaction where they can be committed, subject to the usual constraints of conflict-checking. If you begin a new transaction while already inside a transaction, the effect is the same as an abort.

In manual transaction mode, as in automatic mode, an unsuccessful commit leaves the session in the current transaction until you take steps to end the transaction by aborting.

Choosing Which Mode to Use

You should use automatic transaction mode if the work you are doing requires committing to the repository frequently, because you can make permanent changes to the repository only when you are in a transaction.

Use manual transaction mode if the work you are doing requires looking at objects in the repository, but only seldom requires committing changes to the repository. You will have to start a transaction manually before you can commit your changes to the repository, but the system will be able to run with less overhead.

Switching Between Modes

To find out if you are currently in a transaction, execute *aGbsSession* `inTransaction`. This returns `true` if you are in a transaction and `false` if you are not.

To change from manual to automatic transaction mode, execute the expression:

aGbsSession `transactionMode: #autoBegin`

This message automatically aborts the transaction, if any, changes the transaction mode, and starts a new transaction.

To change from automatic to manual transaction mode, execute the expression:

aGbsSession `transactionMode: #manualBegin`

This message automatically aborts the current transaction and changes the transaction mode to manual. It does not start a new transaction, but it does provide a fresh view of the repository.

7.5 Managing Concurrent Transactions

When you tell GemStone to commit your transaction, it checks to see if doing so presents a conflict with the activities of any other users.

1. It checks to see whether other concurrent sessions have committed transactions of their own, modifying an object that you have also modified during your transaction. If they have, then the resulting modified objects can be inconsistent with each other.
2. It checks to see whether other concurrent sessions have committed transactions of their own, modifying an object that you have read during your transaction, while at the same time you have modified an object that the other session has read.
3. It checks for locks set by other sessions that indicate the intention to modify objects that you have read or to read objects you have modified in your view.

If it finds no such conflicts, GemStone commits the transaction, and your work becomes part of the permanent, shared repository. Your view of the repository is refreshed and any new or modified objects that other users have recently committed become visible in any dictionaries that you share with them.

Read and Write Operations

It is customary to consider the operations that take place within a transaction as *reading* or *writing* objects. Any operation that accesses any instance variable of an object *reads* that object, as do operations that fetch an object's size, class, or other descriptive information about that object. An object also is read in the process of being stored into another object.

An operation that stores a value in one of an object's instance variables *writes* the object. While you can read without writing, writing an object always implies reading it, because GemStone must read the internal state of an object in order to store a value in it.

In order to detect conflict among concurrent users, GemStone maintains two logical sets for each session: a set containing objects read during a transaction and a set containing objects written. These sets are called the *read set* and the *write set*. Because writing implies reading, the read set is always a superset of the write set.

The following conditions signal a possible concurrency conflict:

- An object in your write set is also in another transaction's write set (a *write/write* conflict).
- An object in your write set is in another transaction's read set *and* an object in your read set is in that transaction's write set (a *read/write* conflict).

Optimistic and Pessimistic Concurrency Control

GemStone provides two approaches to managing concurrent transactions: optimistic and pessimistic. An application can use either or both approaches, as needed.

Optimistic concurrency control means that you simply read and write objects as if you were the only session, letting GemStone detect conflicts with other sessions only when you try to commit a transaction.

Pessimistic concurrency control means that you act as early as possible to prevent conflicts by explicitly requesting locks on objects before you modify them. When an object is locked, other users are unable to lock the object or commit changes to it.

Optimistic concurrency control is easy to implement in an application, but you run the risk of having to re-do the work you've done if conflicts are detected and you're unable to commit. When GemStone looks for conflicts only at commit time, your chances of being in conflict with other users increase with the time between commits and the size of your read and write sets. Under optimistic concurrency control, GemStone detects conflict by comparing your read and write sets with those of all other transactions committed since your transaction began.

Running under optimistic concurrency control is the most convenient and efficient mode of operation when:

- you are not sharing data with other sessions, *or*
- you are reading data but not writing, *or*

- you are writing a limited amount of shared data *and* you can tolerate not being able to commit your work sometimes.

If you take a pessimistic approach, you act as early as possible to prevent conflicts by explicitly requesting locks on objects before you modify them. When an object is locked, other people are unable to lock the object, and they cannot optimistically commit changes to the object. Also, when you encounter an object that someone else has locked, you can abort the transaction immediately instead of wasting time on work that can't be committed.

Locking improves one user's chances of committing, but at the expense of other users, so you should use locks sparingly to prevent an overall degradation of system performance. Still, if there is a lot of competition for shared objects in your application, or if you can't tolerate even an occasional inability to commit, then using locks might be your best choice.

Locks do not prevent read-only access to objects, so read-only query transactions are not affected by modification transactions.

Setting the Concurrency Mode

This section applies to GemStone/S only; GemStone/S 64 Bit always uses a Concurrency Mode of FULL_CHECKS.

Concurrency Mode is Any shared object that is not explicitly locked is treated optimistically. For objects under optimistic concurrency control, GemStone's level of checking for concurrency conflicts is configurable. You can set the level of checking for concurrency conflicts by specifying one of the following values for the CONCURRENCY_MODE configuration parameter in your application's configuration file. There are two levels:

- FULL_CHECKS (the default mode), which checks for both *write/write* and *read/write* conflicts. If either type of conflict is detected your transaction cannot commit.
- NO_RW_CHECKS, which performs *write/write* checking only.

Locking methods override the configured optimistic CONCURRENCY_MODE by stating explicitly the kind of pessimistic control they implement.

Setting Locks

GemBuilder provides locking protocol that allows application developers to write client Smalltalk code to lock objects and specify client Smalltalk code to be executed if locking fails.

A GbsSession is the receiver of all lock requests. Locks can be requested on a single object or on a collection of objects. Single lock requests are made with the following statements:

aGbsSession `readLock:`*anObject*`.`
aGbsSession `writeLock:`*anObject*`.`
aGbsSession `exclusiveLock:`*anObject*`.`

The above messages request a particular type of lock on *anObject*. If the lock is granted, the method returns the receiver. (Lock types are described in the *GemStone Programming Guide*; note that exclusive locks are not available in GemStone / S 64 Bit). If you don't have the proper authorization, or if another session already has a conflicting lock, an error will be generated.

When you request an exclusive lock, an error will be generated if another session has committed a change to *anObject* since the beginning of the current transaction. In this case, the lock is granted despite the error, but it is seen as "dirty." A session holding a dirty lock cannot commit its transaction, but must abort to obtain an up-to-date value for *anObject*. The lock will remain, however, after the transaction is aborted.

Another version of the lock request allows these possible error conditions to be detected and acted on.

aGbsSession `readLock:`*anObject* `ifDenied:`*block1* `ifChanged:`*block2*
aGbsSession `writeLock:`*anObject* `ifDenied:`*block1* `ifChanged:`*block2*
aGbsSession `exclusiveLock:`*anObject* `ifDenied:`*block1* `ifChanged:`*block2*

If another session has committed a change to *anObject* since the beginning of the current transaction, the lock is granted but dirty, and the method returns the value of the zero-argument block *block2*.

The following statements request locks on each element in the three different collections.

aGbsSession `readLockAll:`*aCollection*`.`
aGbsSession `writeLockAll:`*aCollection*`.`
aGbsSession `exclusiveLockAll:`*aCollection*`.`

The following statements request locks on a collection, acquiring locks on as many objects in *aCollection* as possible. If you do not have the proper authorization for any object in the collection, an error is generated and no locks are granted.

aGbsSession `readLockAll:` *aCollection* `ifIncomplete:` *block1*
aGbsSession `writeLockAll:` *aCollection* `ifIncomplete:` *block1*
aGbsSession `exclusiveLockAll:` *aCollection* `ifIncomplete:` *block1*

Example 7.1 shows how error handling might be implemented for the collection locking methods:

Example 7.1

```
getWriteLocksOn:aCollection
  "This method attempts to set write locks on the elements
   of a Collection."
aGbsSession
   writeLockAll: aCollection
   ifIncomplete: [ :result |
                  (result at: 1)isEmpty ifFalse:
                        [self handleDenialOn: denied].
                  (result at: 2)isEmpty ifFalse:
                        [aGbsSession abortTransaction].
                  (result at: 3)isEmpty ifFalse:
                        [aGbsSession abortTransaction].
               ].
```

Once you lock an object, it normally remains locked until you either log out or explicitly remove the lock; unless you specify otherwise, locks persist through aborts and commits. In general, you should remove a lock on an object when you have used the object, committed the resulting values to the repository, and no longer anticipate a need to maintain control of the object.

The following methods are used to remove specific locks.

aGbsSession `removeLock:` *anObject*`.`
aGbsSession `removeLockAll:` *aCollection*`.`
aGbsSession `removeLocksForSession.`

The following methods answer various lock inquiries:

aGbsSession `sessionLocks.`
aGbsSession `systemLocks.`
aGbsSession `lockOwners:` *anObject*`.`
aGbsSession `lockKind:` *anObject.*

Releasing Locks Upon Aborting or Committing

The following statements add a locked object or the locked elements of a collection to the set of objects whose locks are to be released upon the next commit or abort:

aGbsSession `addToCommitReleaseLocksSet:` *aLockedObject*
aGbsSession `addToCommitOrAbortReleaseLocksSet:` *aLockedObject*
aGbsSession `addAllToCommitReleaseLocksSet:` *aLockedCollection*
aGbsSession `addAllToCommitOrAbortReleaseLocksSet:` *aLockedCollection*

If you add an object to one of these sets and then request a fresh lock on it, the object is removed from the set.

You can remove objects from these sets without removing the lock on the object. The following statements show how to do this:

aGbsSession `removeFromCommitReleaseLocksSet:` *aLockedObject*
aGbsSession `removeFromCommitOrAbortReleaseLocksSet:` *aLockedObject*
aGbsSession `removeAllFromCommitReleaseLocksSet:` *aLockedCollection*
aGbsSession `removeAllFromCommitOrAbortReleaseLocksSet:` *aLockedCollection*

The following GemStone Smalltalk statements remove all objects from the set of objects whose locks are to be released upon the next commit or abort. These methods are executed using **GS-Do it**

```
System clearCommitReleaseLocksSet
System clearCommitOrAbortReleaseLocksSet
```

The statement *aGbsSession* `commitAndReleaseLocks` attempts to commit the current transaction, and clears all locks for the session if the transaction was successfully committed.

7.6 Reduced-conflict Classes

At times GemStone will perceive a conflict when two users are accessing the same object, when what the users are doing actually presents no problem. For example, GemStone may perceive a write / write conflict when two users are simultaneously trying to add an object to a Bag that they both have access to because this is seen as modifying the Bag.

GemStone provides some reduced-conflict classes that can be used instead of their regular counterparts in applications that might otherwise experience too many unnecessary conflicts. These classes are RcCounter, RcIdentityBag, RcKeyValueDictionary, and RcQueue.

Use of these classes can improve performance by allowing a greater number of transactions to commit successfully without locks, but they do carry some overhead.

For one thing, they use more storage than their ordinary counterparts. Also, you may find that your application takes longer to commit transactions when you use instances of these classes. Finally, you should be aware that under certain circumstances, instances of these classes can hide conflicts from you that you indeed need to know about. Because of the way these classes are implemented, GemBuilder creates instances of these classes as forwarders, rather than replicates.

Here are brief descriptions of the reduced conflict classes. For details about these classes and their usage, see the chapter in the *GemStone Programming Guide* entitled "Transactions and Concurrency Control."

RcCounter
RcCounter maintains an integral value that can be incremented or decremented. A single instance of RcCounter can be shared among multiple concurrent sessions without conflict.

RcIdentityBag
RcIdentityBag provides the same functionality as IdentityBag, except that no conflict occurs on instances of RcIdentityBag when a number of users read objects in the bag or add objects to the bag at the same time. Nor is there a conflict when one user removes an object from the bag while other users are adding objects, or when a number of users remove objects from the bag at the same time, so long as no more than one of them tries to remove the last occurrence of an object.

RcKeyValueDictionary
This class provides the same functionality as KeyValueDictionary except that no conflict occurs on instances of RcKeyValueDictionary when users read

values in the dictionary or add keys and values to it (unless one tries to add a key that already exists) or when users remove keys from the dictionary at the same time (unless more than one user tries to remove the same key at the same time).

Conflict occurs only when more than one user tries to modify or remove the same key from the dictionary at the same time.

RcQueue

The class RcQueue represents a first-in-first-out (FIFO) queue. No conflict occurs on instances of RcQueue when multiple users read objects in or add objects to the queue at the same time, or when one user removes an object from the queue while other users are adding objects. However, if more than one user removes objects from the queue, they are likely to experience a write / write conflict.

7.7 Changed Object Notification

A *notifier* is an optional signal that is activated when an object's committed state changes. Notifiers allow sessions to monitor the status of designated shared application objects. A program that monitors stock prices, for example, could use notifiers to detect changes in the prices of certain stocks.

In order to be notified that an object has changed, a session must register that object with the system by adding it to the session's *notify set*.

Notify sets are virtual but persist through transactions, living as long as the GemStone session in which they were created. When the session ends, the notify set is no longer in effect. If you need it for your next session, you must recreate it. However, you need not recreate it from one transaction to the next.

Class GbsSession provides the following two methods for adding objects to notifySets:

`addToNotifySet:`
: adds one object to the notify set

`addAllToNotifySet:`
: adds the contents of a collection to the notify set

When an object in the notify set appears in the write set of any committing transaction, the system executes a previously defined client Smalltalk block, sending a collection of the objects signaled as its argument. By examining the argument, the session can determine exactly which object triggered the signal.

Because these events are not initiated by your session but cause code to run within your session, this code is run asynchronously in a separate Smalltalk process. Depending on what else is occurring in your application at that time, using this feature might introduce multi-threading into your application, requiring you to take some additional precautions. (See “Multiprocess Applications” on page 209.)

Example 7.2 demonstrates notification in GemBuilder.

Example 7.2

```
"First, set up notifying objects and notification action"
| notifier |
GBSM currentSession abortTransaction; clearNotifySet.
notifier := Array new: 1.
GBSM currentSession at: #Notifier put: notifier.
GBSM currentSession commitTransaction.
GBSM currentSession addToNotifySet: notifier.
GBSM currentSession notificationAction: [ :objs |
      Transcript cr; show: 'Notification received' ]

"Now, from any session logged into the same stone with
visibility to the object 'notifier' - to initiate
notification"
GBSM currentSession abortTransaction;
      evaluate: 'Notifier at: 1 put: Object new';
      commitTransaction
```

7.8 Gem-to-Gem Notification

Sessions can send general purpose signals to other GemStone sessions, allowing the transmission of the sender’s session, a numerical signal value, and an associated message.

One Gem can handle a signal from another using the method GbsSession >> `sessionSignalAction:` *aBlock*, where *aBlock* is a one-argument block that will be passed a forwarder to the instance of GsInterSessionSignal that was received. From the GsInterSessionSignal instance, you can extract the session, signal value, and string.

One GemStone session sends a signal to another with:

aGbsSession `sendSignal:` *aSignal* `to:` *aSessionId* `withMessage:` *aString*

For example:

Example 7.3

```
"First, set up the signal-receiving action"
GBSM currentSession sessionSignalAction: [ :giss |
       nil gbsMessenger
              comment: 'Signal %1 received from session %2: %3.'
              with: giss signal
              with: giss session
              with: giss message.
].

"Now, from any session logged into the same Stone, send a
signal.(This example uses the same session)"
GBSM currentSession
       sendSignal: 15
       to: (GBSM evaluate: 'GsCurrentSession currentSession serialNumber')
       withMessage: 'This is the signal'.
```

If the signal is received during GemStone execution, the signal is processed and execution continues. If *aBlock* is `nil`, any previously installed signal action is deinstalled.

NOTE

The method `sessionSignalAction:` *and the mechanism described above supersede the old mechanism that used the method* `gemSignalAction:`. *Do not use both this method and* `gemSignalAction:` *during the same session; only the last defined signal handler will remain in effect.*

See the chapter entitled “Error-handling” in your *GemStone Programming Guide* for details on using the error mechanism for change notification.

7.9 Asynchronous Event Error Handling

For each sesion, there is a background thread that detects events from the server such as sigAbort, lostOTroot, gem to gem signals, and changed object notifications, and other events that are handled internally. If a non-fatal error occurs in processing these events, by default a walkback is opened.

To avoid an end-user experiencing a walkback, you may set a handler block for an unexpected error in this event detection.

```
GbsSession >> eventDetectorErrorHandler: aOneArgBlock
```

If the eventDetectorErrorHandler is set, and if the exception is not already handled by another handler that is set up for the application, this handler block will be executed for the exception caught by the event detection thread.

Chapter

8 Security and Object Access

Once objects have been successfully committed to GemStone, they can be damaged or destroyed only by mishaps that damage or erase the disk files containing your repository. GemStone provides several mechanisms for safeguarding the objects in your GemStone repository. These mechanisms are discussed in the chapter on creating and restoring backups in the *GemStone System Administration Guide*.

This chapter discusses security and access at the object level. Not all these security features are available for all server product versions - GemStone/S 64 Bit 1.0 and 2.0 do not support Segments. Much of the material in the chapter will not apply when using GBS with a GemStone/S 64 Bit server.

GemStone Security
highlights the mechanisms GemStone provides for keeping your stored objects secure.

GemStone Administration Tools
describes the visual tools that you can use to manage access to objects by multiple users: the Segment Tool, the Symbol List Browser, and the User Account Management Tools.

8.1 GemStone Security

GemStone provides for blocking access to certain objects as well as sharing them. Applications can take advantage of several security mechanisms to prevent unauthorized access to, or modification of, sensitive code and data. These mechanisms are listed below, and you can choose to use any or all of them.

GemStone provides security at several levels:

- Login authorization keeps unauthorized users from gaining access to the repository;
- Privileges limit ability to execute special methods affecting the basic functioning of the system; and
- Object level security allows specific groups of users access to individual objects in the repository.

Complete details on GemStone security mechanisms are found in the *GemStone/S System Administration Guide* for your GemStone server product and version. A brief overview is included here.

Login Authorization

GemStone's first line of protection is to control login authorization. When someone tries to log in to GemStone, GemStone requires a user name and password. If the user name and password match the user name and password of someone authorized to use the system, GemStone allows interaction to proceed; if not, the connection is severed.

The GemStone system administrator controls login authorization by establishing user names and passwords when he or she creates UserProfiles.

The UserProfile

Each instance of UserProfile is created by the system administrator. The UserProfile contains information about you as an individual user, such the UserId and password, your SymbolList, any groups you belong to, and your privileges. This information is used to provide system and object level security, including object visibility.

Controlling Visibility of Objects with SymbolLists

One way to control access is to hide certain objects from users. Each GemStone user has a SymbolList, containing a collection of SymbolDictionaries to which they

have been given access. Objects—such as Classes—that are not found in a search of the user's SymbolLists are not accessible. Because it is difficult for users to refer to objects that are not defined somewhere in their symbol lists, simply omitting off-limits objects from a user's symbol list provides a certain amount of security. It is possible, however, for users to find ways to circumvent this, since it's difficult to ensure that all indirect paths to an object are eliminated.

NOTE

For performance reasons, GbsSession uses transient copies of your symbol lists. If you change this transient copy programmatically, the changes are not immediately reflected in the permanent GemStone object. Also, changes to the permanent GemStone symbol list are not reflected in the GbsSession's transient symbol list until a transaction boundary. If you must be absolutely certain that the two copies are synchronized, log out and log back in again.

System Privileges

A few GemStone Smalltalk methods can be executed only by those who have explicitly been given the necessary *privileges.* The privilege mechanism is entirely independent of the authorization mechanism. This mechanism allows the system administrator to control who can send certain powerful messages, such as those that halt the system or change passwords. Privileges are associated with only certain methods and cannot be extended to others.

Specific privileges and the privileged messages are described in the image, and their use is discussed in the *GemStone System Administration Guide.*

Protecting Methods

Another choice is to implement procedural protection. If your program accesses its objects only through methods, you can control the use of those objects by including user identity checks in the accessing methods.

Object-level Security

Segments

Instances of GemStone's Segment Class provide read and write authorization control to individual objects. A Segment is an object access policy. When someone tries to read or write an object that is associated with a Segment to which he or she lacks the proper authorization, GemStone raises an authorization error and does not permit the requested operation.

In GemStone/S, every server object is associated with an instance of Segment that controls access to that object. In GemStone/S 64 Bit v2.2 and later, objects may be associated with a Segment or not. If not, no object authorization is done and any user can read and write the objects.

All objects associated with a particular segment have exactly the same protection; that is, if you can read or write one object with that segment, you can read or write them all. Each segment is owned by a specific single user, and may have authorizations for owner, groups, or world for read-only, read-write, or no access.

Groups provide a way to allow a number of GemStone users to share the same level of access to set of objects in the repository.

Segments are not meant to organize objects for retrieval; GemStone uses Symbol Lists for that. Moreover, segments don't have any relationship to the physical location of objects on disk; they merely provide access security.

For a complete discussion of symbol resolution and object sharing, see the relevant chapters of the *GemStone Programming Guide*.

8.2 GemStone Administration Tools

The following sections describe the GemStone tools that are provided to allow you to easily manage the object sharing and protection issues discussed in the previous section.

- The **Segment Tool** is a tool for examining and changing GemStone user authorization.
- The **Symbol List Browser** is a tool you can use for examining the GemStone SymbolLists associated with UserProfiles. You can use it to add and delete dictionaries from these lists, as well as to add, delete and inspect the entries in the dictionaries.
- The **User Account Management Tools** are a set of three tools, the **User List**, the **User Dialog**, and the **Privileges Dialog**, that allow you to create new user accounts, change account passwords, and assign group memberships.

The Segment Tool

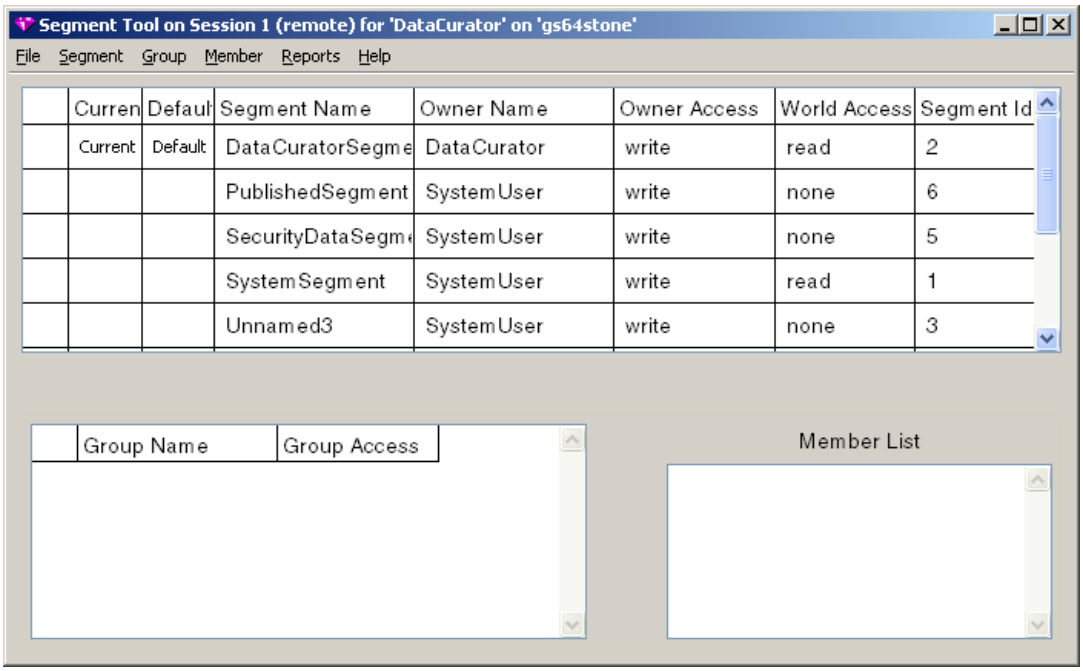
The Segment Tool allows you to inspect and change the authorization that GemStone users have at the object level. As explained in the section entitled "Object-level Security" beginning on page 163, each object in GemStone is assigned separately to a segment. The only users authorized to read or modify an

object are those who are granted read or write authorization by the segment it is assigned to. The Segment Tool also allows you to examine and change group membership.

Some of the operations supported by the Segment Tool involve privileged methods. If your account does not have the needed privileges, ask your system administrator to set up your segments for you.

To open a Segment Tool, select **Admin > Segments** from the GemStone menu or through the User Dialog's **Show Segments** button. Figure 8.1 shows a Segment Tool.

Figure 8.1 The Segment Tool


Segment Tool on Session 1 (remote) for 'DataCurator' on 'gs64stone'
File Segment Group Member Reports Help
Curren Defaul Segment Name Owner Name Owner Access World Access Segment Id
Current Default DataCuratorSegme DataCurator write read 2
PublishedSegment SystemUser write none 6
SecurityDataSegm SystemUser write none 5
SystemSegment SystemUser write read 1
Unnamed3 SystemUser write none 3
Group Name Group Access
Member List

The Segment Tool is divided into two main sections: the upper section displays segments; the lower section displays groups and their members.

Segment Definition Area

The segment definition area at the top of the dialog displays the segments in the SystemRepository to which the current user has read authorization.

You will notice that some segments are named and some are unnamed. Named segments are segments that are referenced in a dictionary or symbol list; unnamed segments are those that are not referenced in any dictionary or symbol list

In addition to the segments displayed in the Segment Tool, all users also have read and write authorization to GsIndexingSegment. Because authorization changes should not be made to that segment, however, it is not included in the tool.

NOTE

Changes made to cells in the tables are accepted automatically as soon as you either press Return, make a selection in a combo box associated with the cell, or simply move the focus to another cell or field by moving the mouse. If you enter an invalid value in a cell results in a warning, and the cell reverts to the original value.

In the segment definition area of the Tool you can change the following:

Current — You can set the segment to be the current segment: the segment to which GemStone will assign any objects you create now.

Default — You can set the segment to be the default segment: the home segment that is your current segment when you log in to GemStone.

Owner Name — You can enter any valid user name that already exists in the system. To change an owner name, type a valid owner name into the cell.

Owner Access and **World Access —** To change owner and world access, type one of the following values into their cells:

- **read** means that a user can read any of the segment's objects, but can't modify (write) them or add new ones
- **write** means that a user can read and modify any of the segment's objects and create new objects associated with the segment
- **none** means that a user can neither read nor write any of the segment's objects

NOTE

Be careful when changing the authorizations on any segment that a user may be using as a current segment or a default segment.

Segment Id — The Id number of each Segment is displayed. This information can not be modified.

Group Definition Area

The bottom of the dialog is the group definition area. In this area you can assign authorizations to groups of users instead of individuals. Groups are typically organized as categories of users who have common interests or needs.

When you select a segment at the top of the dialog, the group definition area displays the groups that have access to the segment. When you select one of the groups, its members appear.

In the group definition area you can change the following:

Group Name — You can change the group name, but you should be aware that when you edit a group name, you are not just renaming the group; you are actually replacing the group with a new one. The old group's members are not copied to the new one, so you need to add them again. If the name of the group entered is a group that does not exist, you will be asked if you want to create it.

Group Access — Group access can be changed in the same way as owner and world access. To change group access, type either **read** or **write** into the cell, as outlined for owner and world access on page 166.

NOTE

Be careful when changing the authorizations on any segment that a user may be using as a current segment or a default segment.

If you want to add group access to a segment, select **add...** from the pop-up menu in a Group Name cell. Similarly, to remove group access from a segment, select **remove...** from the pop-up menu.

In addition, you can select groups and users here to be the receiver of actions on the menus.

Segment Tool Menus

The following sections describe the menus that are available in the Segment Tool.

The File Menu

Use the **File** menu to commit work done in the Segment Tool, to abort the transaction, to update the tool's view of segments, groups, and users in the current session, and to close the Segment Tool.

Table 8.1 File Menu in the Segment Tool

Commit	Commits all the work executed in GemStone during the current transaction. After you commit, you are given a new, updated view of the repository, and you can continue your work.
Abort...	Cancels all changes that you have made anywhere in GemStone since your last commit. After you abort the transaction, you are given a new, updated view of the repository, and you can continue your work.
Update	Updates the information in the Segment Tool and gives you a new, updated view of segments, groups, and users that reflects the most recent version of the repository, and you can continue your work.
Close	Close the Segment tool. Any changes made are kept, but not committed to the repository.

Segment Menu

Use the **Segment** menu to create new segments and to manipulate existing segments.

Table 8.2 Segment Menu in the Segment Tool

Create...	Creates a new segment. You must have the SegmentCreation privilege to use this option. In the Create Segment dialog, enter a name for the segment and a symbol dictionary to store it in. Private segments are typically kept in UserGlobals. Segments for large groups of users are typically kept in Globals.
Grab	Grabs a reference to the selected segment and places it on the clipboard. This can be used to add a reference to a user's symbol list or for changing the default segment of a user in the User Dialog.
Make Current	Makes the selected segment your current segment. When you create an object, GemStone assigns it to your current segment.
Make Default	Makes the selected segment your default segment. This is the home segment that is your current segment when you log into GemStone.

Group Menu

Use the **Group** menu to add and remove groups.

Table 8.3 Group Menu in the Segment Tool

Add...	Adds a new group. In the Add Group dialog, enter a name for the group and choose **OK** or **Apply**.
Remove...	Removes authorization for the selected group. This does not delete the group from GemStone. It only means that the current segment no longer stores access information for that group. Users may still be able to access other objects because of their membership in the group, but they will not have access to the objects assigned to this segment unless it has been provided by the segment's owner or world access.

Member Menu

Use the **Member** menu to add users to and remove users from groups.

Table 8.4 Member Menu in the Segment Tool

Add...	Adds a user to the group. Enter any valid user name in the Add Member dialog and choose **OK** or **Apply**. The user must already exist in the system. You can use the User List to create new users.
Remove...	Removes the selected user from the group. (This does not delete the user from GemStone.)

Reports Menu

Use the **Reports** menu to bring up a window displaying information about the segments, users, and groups in your view of the repository. Use the window's **Print** button to print a report, and use the **Cancel** button to close the window.

Table 8.5 Report Menu in the Segment Tool

Group Report	Produces a list of all groups in GemStone and the users in each group.
Segment Report	Produces a list of segments the user has read authorization for and displays information about each one as to • its owner, • the groups for which it contains access information, and • the access it grants to the owner, groups, and world. This report includes the GsIndexingSegment, for which all users have read and write authorization.
User Report	Produces a list of all GemStone users and shows each user's group memberships.

Segments that appear as Unnamed are not in your symbol list. Thus, their names and dictionaries are unknown.

Help Menu

The **Help** menu contains one item, **Session Info**, which provides information about the session for the Segment Tool window and about the current session.

Using the Segment Tool

If you are a segment's owner, you can determine who has access to objects assigned to that segment. For more information, see the chapter on administering user accounts and segments in the *GemStone System Administration Guide.*

Checking Segment Authorization

Anyone who has read authorization for a segment can use the Segment Tool to find out who is authorized to read or write that segment by doing the following:

1. Bring up the Segment Tool by selecting **GemStone > Admin > Segments** or by choosing **Show Segments** in a GemStone User Dialog.
2. In the Segment Tool, choose **Reports > Segment Report**. The resulting list contains all segments.
3. To view the members of each group, choose **Reports > Group Report**. To view the groups to which each user belongs, choose **Reports > User Report**.

Changing Segment Authorization

Assuming that you either have SegmentProtection privileges or are the segment's owner, you can use the Segment Tool to change the authorization of a segment.

The top half of the Segment Tool shows the owner, the owner's access, and world access for each segment in the repository. To change owner or world access, select the existing permission you want to change. Then enter a new permission ("read", "write", or "none").

The new authorization will take effect when you commit the current transaction.

CAUTION

Be careful to check whether a user is logged in before you remove write authorization. A user will be unable to commit changes if write authorization is removed from the current segment, and if it is the user's default segment, the user's session will be terminated and the user will be unable to log in again.

Controlling Group Access to a Segment

If you are authorized to set up or change group access to a segment, you can add or remove groups to that segment's authorization list.

- Make sure the segment is selected in the top half of the tool.
- To add a group to the authorization list for the selected segment, choose **Add...** from the **Group** menu. Enter the group name in the dialog box that appears. If the group does not exist in the repository, you will be asked whether to create it.

- To remove a group from the authorization list, first select the group by clicking in the first column of the groups list. Then choose **Remove...** from the **Group** menu. You will be asked to confirm the action.
- To change the type of access for a particular group, first select that group in the groups list and select the existing permission. Then enter the new permission ("read" or "write").
- To add a member to a group that has access to this segment, first select that group in the groups list. Then choose **Add...** from the **Member** menu. Enter the UserId and choose **OK**. (A UserProfile with that UserId must already exist in the repository.)
- To remove a member from a group that has access to this segment, select the UserId in the member list and choose **Remove...** from the **Member** menu. You will be asked to confirm the action.

Remember to commit your transaction before logging out. A convenient way to do that is by choosing **Commit** from this tool's **File** menu.

Changing a User's Default Segment

You must either have DefaultSegment privileges to change your own default segment, or have write authorization in the DataCurator Segment to change another user's default segment.

To change your own default segment, select the desired segment by clicking in its row. Then choose **Segment > Make Default**.

To change someone else's default segment, in the GemStone User Dialog, choose **Show Segments**.

1. In the Segment Tool, select the desired segment by clicking in its row. Choose **Segment > Grab**.
2. In the GemStone User Dialog, choose **Paste To Default Segment**.
3. Choose **OK** or **Apply**.

NOTE

Changes to a segment's authorization do not take effect until the current transaction is committed. This means that if you change any user's default segment (including your own) to a segment for which that user lacks write authorization, and you subsequently commit the transaction, the affected user will no longer be able to log in to GemStone.

The Symbol List Browser

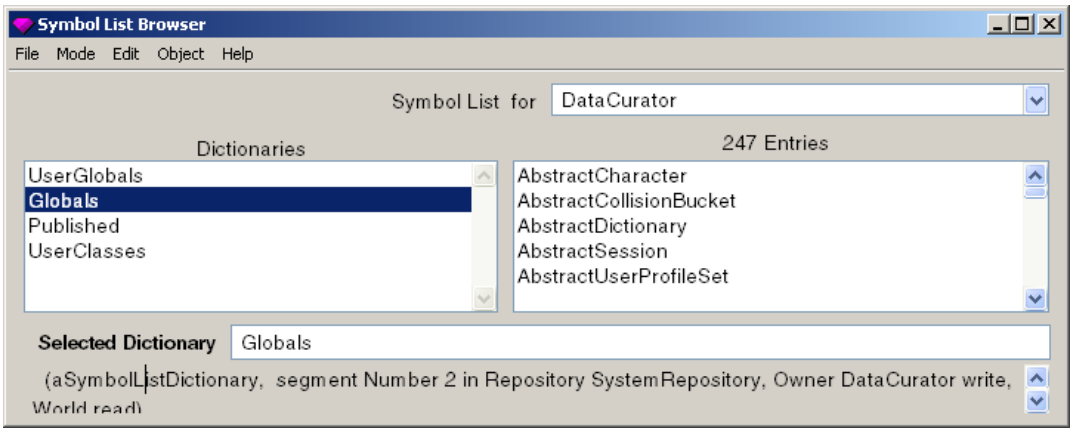
The Symbol List Browser is a tool for examining the GemStone SymbolLists associated with UserProfiles, adding and deleting dictionaries from these lists, examining the entries in those dictionaries and adding, deleting and inspecting the entries. References to dictionaries and dictionary entries can be copied between GemStone user accounts, subject to authorization and segment restrictions, to allow users to share application objects and name spaces developed by other users, and to publish them to other users.

To open a Symbol List Browser, select **Admin > Symbol Lists** from the Gemstone menu, or click on the **Show Symbol List** button on a GemStone User Dialog.

Like the other GemStone tools, the Symbol List Browser opens on a particular login session. When a Symbol List Browser instance is created, it is attached to the current GemStone session and remains attached to that session until the browser is closed.

Figure 8.2 shows the Symbol List Browser.

Figure 8.2 The Symbol List Browser



The field labeled **Symbol List for** contains a list of all the GemStone users that are visible to the session to which the browser is attached. When you select a GemStone user name, a list of the dictionaries in that user's SymbolList is displayed in the **Dictionaries** pane. GemStone permissions are observed; any

dictionaries in that SymbolList that are not normally accessible to the browser's session will not be visible in the list.

When a dictionary is selected, the keys of the entries in the dictionary are displayed in the **Entries** pane on the right.

Whenever a dictionary or an entry is selected, information about that object is displayed at the bottom of the browser.

The Clipboard

Within the Symbol List Browser you can delete, move, and copy objects to and from SymbolLists and the Dictionaries in those SymbolLists. For each session to which a Symbol List Browser is attached, there is a "clipboard" onto which GemStone server objects can be cut and copied and from which objects can be pasted into another Symbol List Browser that is also attached to that session.

Symbol List Browser Menus

The menus in the symbol list browser allow you to examine, add, and delete SymbolLists, dictionaries, and dictionary entries. You can use this browser to copy references to dictionaries and dictionary entries among user accounts so application objects can be shared by other users.

File Menu

The **File** menu contains items for operating on the window itself and for committing and aborting transactions from the Symbol List Browser.

Table 8.6 File Menu in the Symbol List Browser

Commit	Makes all changes in the current transaction permanent.
Abort	Aborts the current transactions.
Update	Updates the browser's view of the GemStone server objects it shows. The browser is automatically updated if the attached session aborts a transaction.
Close	Close the browser. Any changes are retained in the image, but not committed to the repository.

Mode Menu

The Mode menu allows you to switch from dictionary mode to entry mode. In dictionary mode, you can select entries and dictionaries from the lists. In entry mode, you can edit or enter new text in the Symbol List and Selected Entry fields.

Edit Menu

In Dictionary Mode, the Edit menu allows you to rearrange dictionaries by cutting, copying, or pasting. In Entry Mode, the Edit menu allows you to rearrange entries by cutting, copying, or pasting.

Table 8.7 Edit Menu in the Symbol List Browser

Cut Dict **Cut Entry**	*In Dictionary mode:* Removes the selected dictionary from the user's symbol list and places it in the session's clipboard. *In Entry mode:* Removes the selected entry from the selected Dictionary and places it in the session's clipboard.
Copy Dict **Copy Entry**	Copies a reference to the selected item (a dictionary or an entry, depending which mode is in effect) into the session's clipboard.
Paste Dict **Paste Entry**	*In Dictionary mode:* Causes the reference to the dictionary object in the clipboard to be added to the SymbolList in the pane, with the name it had when it was put in the clipboard. *In Entry mode:* Causes the reference to the entry in the clipboard to be added to the selected dictionary, with the name it had when it was put in the clipboard. *In both modes:* If the clipboard item's name is already in use in the destination list, a Confirmer will pop up to allow replacing the old item, or to abort the paste operation.

Object Menu

The Object Menu allows you add a new dictionary, open an inspector on a dictionary entry, and open a browser on a class that is contained in a dictionary. Its menu items are: **Add Dict, Inspect Dict,** and **Browse Class.**

Table 8.8 Object Menu in the Symbol List Browser

Add Dict **Add Entry**	Prompts for name of a new item to be added to the Dictionary or Entry list.
Inspect Dict **Inspect Entry**	Opens a GemStone inspector on the selected item.
Browse Class	If the selected entry is a class, opens a GemStone class browser on that entry. Performs the same operation in either Dictionary or Entry mode.

Help Menu

The **Help** menu contains one item, **Session Info**, which provides information about the session for the Symbol List Browser and about the current session.

User Account Management Tools

The User Account Management tools are a set of three tools that allow the GemStone System Administrator to create new user accounts, change account passwords, and assign group membership.

This section describes the three User Account Management tools: the GemStone User List, the GemStone User Dialog, and the Privileges Dialog.

Note that you must either be DataCurator or have certain privileges to perform most of the system administration functions described in this section. If you are responsible for GemStone system administration, you should refer the chapter on administering user accounts and segments in the *GemStone System Administration Guide* for specific information on user account management. That chapter discusses the privileges you need to manage user accounts and explains how to add and remove users, set up user environments, change passwords and user privileges, and how to add and remove users from groups.

GemStone User List

The GemStone User List window contains a list of all user accounts known to the current repository. The administrator can use this window to delete users and as a starting point to add new users and to change the attributes of GemStone users.

To bring up the GemStone User List, from the **GemStone** menu select **Admin > Users**. Figure 8.3 shows the GemStone User List.

Figure 8.3 GemStone User List

The GemStone Users List window has two menus: **File** and **Help**.

The **File** menu contains the following items:

Table 8.9 GemStone User List: File Menu

Commit	Makes all changes in the current transaction permanent.
Abort	Aborts the current transaction.
Update	Causes the browser to update its view of the GemStone users it shows. The browser will automatically be updated if the attached session aborts a transaction.
Close	Close the browser. Changes are retained in the image, but not committed to the repository.

Table 8.10 shows operations available in this dialog.

Table 8.10 GemStone User List

Create User...	Brings up a GemStone User dialog in which you can define a new user.
Show User Info...	Brings up a GemStone User window displaying privilege and group membership information on a selected user.
Delete User...	Allows you to remove a user name.

The **Help** menu contains one item, **Session Info**, which provides information about the session for the GemStone User List and about the current session.

GemStone User Dialog

The GemStone User Dialog displays the attributes of a particular GemStone user. The GemStone administrator can examine and change the user's privileges or default segment and can control the user's group membership. The administrator can also change the name available in the user's symbol list.

The GemStone User Dialog is shown in Figure 8.4.

Figure 8.4 GemStone User Dialog

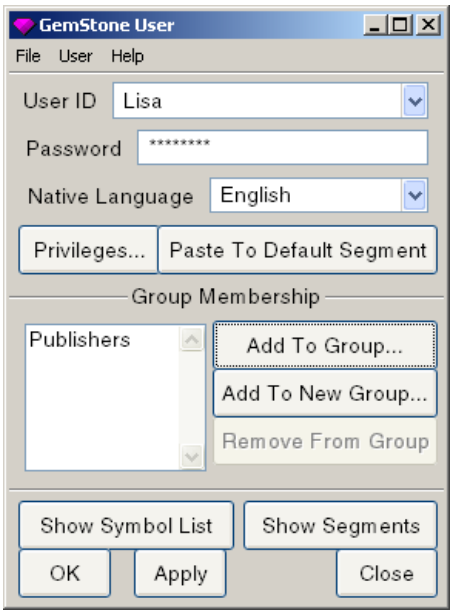


Table 8.11 shows the operations that are available in this dialog.

Table 8.11 Buttons in the GemStone User Dialog

Privileges...	Brings up a Privileges Dialog, in which you can select privileges for this user.
Paste To Default Segment	Makes the grabbed segment from the Segment Tool the default segment for this user.
Add to Group	Allows you to select a group name from a menu of known groups and adds this user to that group.
Add To New Group	Prompts you for a new group name and adds this user to the group.
Remove From Group	Removes this user from the selected group.
Show Symbol List	Brings up a Symbol List Browser for the designated user.
Show Segments	Brings up a Segment Tool.

The User Dialog has three menus, **File**, **User**, and **Help** menus. The **File** menu contains the following items.

Table 8.12 GemStone User Dialog: File Menu

Commit	Makes all changes in the current transaction permanent.
Abort	Aborts the current transaction.
Update	Causes the dialog to update its view of the GemStone user it shows. The dialog will automatically be updated if the attached session aborts a transaction.
Close	Close the dialog. Changes are retained in the image, but not committed to the repository.

The **User** menu contains one item, **Rename**. This requests a new name for this user. You may not rename the DataCurator, GcUser, or SystemUser accounts.

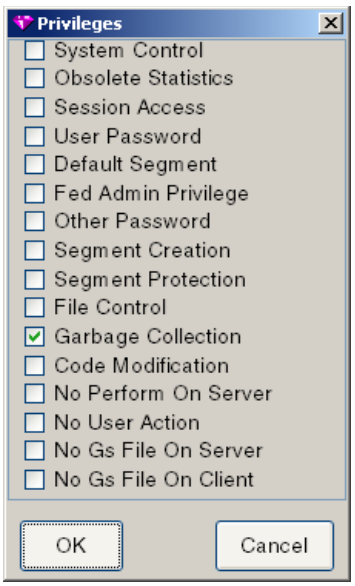
The **Help** menu contains one item, **Session Info**, which provides information about the session for the GemStone User List and about the current session.

The Privileges Dialog

Certain system functions are customarily performed by the DataCurator; for example, many of the messages to System require explicit privilege to use. The privileges dialog displays the privileges an individual user possesses. You can use this dialog to examine a user's privileges, and—if you have the authority to do so—to select privileges for a user. For more information on privileges, see the chapter on Administering User accounts and Segments in the *GemStone System Administration Guide*.

The Privileges Dialog is shown in Figure 8.5.

Figure 8.5 Privileges Dialog in GemStone User Window



The specific list of privileges and the methods associated with these privileges are different in different server products and versions. See Chapter 5 of the *GemStone/S System Administration Guide* for your server product and version for details on the privileges available in GemStone server.

Chapter

9 Exception Handling

This chapter discusses exceptions: how to handle them and how to recover from them, and how to define your own GemStone errors.

GemStone Errors and Exception Classes
describes how GbsErrors and Exceptions are created and used.

Handling Exceptions
explains how to handle exceptions, and how to define and signal your own errors.

Interrupting GemStone Execution
explains how to interrupt GemStone Execution.

9.1 GemStone Errors and Exception Classes

When an error occurs in the GemStone server, an instance of GbsError is created that contains detailed information about the error, and an exception is raised on that GbsError in GemBuilder. You may set up exception handlers to catch the exception, and perform the desired client Smalltalk exception handling. If no exception handler is set up for the particular exception that occurred, the default handler opens a notifier, from which you can open a debugger.

GBS and GemStone server exceptions are represented in client Smalltalk by exception classes. These classes can be found in the application / package GbsExceptions. In VisualWorks, these exception classes are defined in the namespace GemStone.Gbs.

On the GemStone server, there is a dictionary of error names, called ErrorSymbols. This dictionary lists all the errors that can occur when communicating with a GemStone session. For each of these errors, there is a corresponding client exception class. The name of that class is derived by making the first character of the error symbol uppercase, and prepending "Gbs". So, for instance, the server error `#rtErrBadArgKind` corresponds to the client exception class GbsRtErrBadArgKind.

The GemBuilder exception classes fit into the VisualWorks exception hierarchy as shown below:

```
GenericException
   ControlInterrupt
      GbxAbstractControlInterrupt
         <various specific error classes related to pauses, breaks, or breakpoints>
   Exception
      Error
         GbxAbstractException
            GbsGemStoneError
               GbsAbortingError
                  <various specific error classes that cause aborts>
               GbsCompilerError
                  <various specific compile-related error classes>
               GbsEventError
                  <various specific signal related error classes>
               GbsFatalError
                  <various specific fatal error classes>
               GbsInterpreterError
                  <various specific other error classes>
            GbsInterfaceError
               GbsAssertionError
               GbsBlockReplicationError
               GbsClassGenerationError
               GbsConnectorError
               GbsLinkedLoginError
               GbsNotCachedError
               GbsUnsupportedFloatError
               GsiUnsupportedFloatError
```

The subclasses of **GbxAbstractControlInterrupt** are exceptions raised by the GemStone server that are normally used to invoke a debugger. Applications do not typically define handlers for these exceptions.

The subclasses of **GbsGemStoneError** are exceptions that are raised by the GemStone server.

The subclasses of **GbsInterfaceError** represent client side only GBS errors, that is, errors that are detected by the GBS client.

9.2 Handling Exceptions

You can use the on:do: method to install error handlers to anticipate specific GemStone errors. For example, this shows how to use the exception classes to handle a GemStone server error:

Example 9.1

```
[ GBSM currentSession evaluate: '5 / 0' ]
    on: GbsNumErrIntDivisionByZero
    do: [ :ex | ex proceedWith: 'oops' ]
```

To handle an entire set of errors, rather than an individual error, set up the handler for the common superclass of the errors you want to handle. For example, to handle all GemStone fatal errors, set up a handler for GbsFatalError, something like this:

```
[ … ]
    on: GbsFatalError
    do: [ :ex | … ]
```

Each exception detected by the GemStone server also has an associated instance of the class GbsError. This is primarily for internal use by GemBuilder, but you can examine the GbsError by sending #originator to the exception instance. For example:

Example 9.2

```
[ GBSM evaluate: '#( 1 2 3 ) at: 4']
    on: GbsObjErrBadOffsetIncomplete
    do: [ :ex | ex originator inspect ]
```

User-defined Errors

You can define and signal your own errors in GemStone. For more information on how to do this, see the *GemStone Programming Guide.*

In a GemBuilder application, if you want to define a client Smalltalk exception handler for a user-defined error, you will first need to associate the GemStone error number with a client Smalltalk exception class. To do this, use the method `GbsError class>>defineErrorNumber:class:`. This only needs to be done once for each user defined error.

Note that user-defined error numbers must be unique, and the numbers 1000-6999 are reserved for use by GemStone.

For example, suppose you have created a GemStone user-defined error as follows:

Example 9.3

```
"In GemStone"
| myErrors |
myErrors := LanguageDictionary new.
UserGlobals at: #MyErrors put: myErrors.
myErrors at: #English put: (Array new: 10).
(myErrors at: #English)
   at: 10
   put: #( 'My new error with argument ' 1 ).
```

The following client Smalltalk code signals your newly created error in GemStone:

```
GBSM evaluate: 'System signal: 10
   args: #[ 46 ] signalDictionary: MyErrors'
```

A generic signal-handler for all GemStone errors would trap this signal. This code sets up the exception handler and causes the exception to be signaled:

```
^[GBSM evaluate: 'System signal: 10
         args: #[ 46 ]
         signalDictionary: MyErrors']
   on: GbsGemStoneError
   do: [ :ex | ex return: #handled ].
```

To explicitly handle your new error in client Smalltalk, you first need to associate the GemStone error number with a client exception class. Create a new class, which should inherit from GbsGemStoneError. In this example, the class MyNewError

has been created, and this code associates this class with the GemStone error number:

```
GbsError
   defineErrorNumber: 10
   class: MyNewError.
```

Then to explicitly handle your new error from client Smalltalk:

Example 9.4

```
^[ GBSM evaluate: 'System
signal: 10
args: #[ 46 ]
signalDictionary: MyErrors' ]
   on: MyNewError
   do: [ :ex | ex return: #handled ]
```

You can obtain the exception's error description string by sending it #description. For example:

```
[GBSM execute: '#a at: 2']
   on: GbsObjErrBadOffsetIncomplete
   do: [ :ex | ex return: ex description ]
```

You can obtain the array of server exception arguments by sending #serverArguments to the client exception. This array contains client replicates of the server error arguments.

For example:

```
[GBSM execute: '#a at: 2']
   on: GbsObjErrBadOffsetIncomplete
   do: [ :ex | ex return: ex serverArguments ]
```

For information on how to create GemStone error dictionaries and how to handle GemStone errors (predefined and user-defined) within the GemStone environment, see the chapter entitled “Handling Errors” in the GemStone Programming Guide. For more information about defining error handlers in the client Smalltalk, refer to your client Smalltalk documentation on exception handling.

9.3 Interrupting GemStone Execution

When executing GemStone server Smalltalk in a non-blocking RPC session, it's possible to interrupt GemStone execution by sending the session #softBreak. GemStone responds by raising GbsRtErrSoftBreak, which is a continuable exception.

GBS automatically sends #softBreak when a client Smalltalk user interrupt occurs in a Process executing GemStone server code (in a non-blocking RPC session). If VisualWorks can't determine which Process to interrupt when control-Y is entered, try executing GBSM softBreak, or aGbsSession softBreak (if you have more than one GemStone session).

In addition to being continuable, soft break can also be handled on the server by an exception handler. There is another mechanism called hard break which is not continuable, and cannot be handled on the server. Sending an executing GbsSession the message #hardBreak stops server execution, aborts the current GemStone transaction, and raises GbsRtErrHardBreak on the client.

In GBS, in situations where a client Smalltalk user interrupt sends a soft break, a hard break will be sent after the server has not responded to three soft breaks.

Chapter

10 Schema Modification and Coordination

No matter how elegantly your schema was designed, sooner or later changes in your application requirements or even changes in the world around your application will probably make it necessary to make changes to classes that are already instantiated and in use. When this happens, you will want the process of propagating your changes to be smooth and to impact your work as little as possible.

This chapter discusses the mechanisms GemStone and GemBuilder provide to help you accomplish this.

Schema Modification
explains how GemStone supports schema modification by maintaining versions of classes in class history objects. It shows you how to migrate some or all instances from one version of a class to another while retaining the data that these instances hold.

Schema Coordination
explains how to synchronize schema modifications between GemStone and the client Smalltalk.

The Class Version Browser
describes a specialized Class Browser that can be used for examining a class

history, inspecting instances, migrating instances, deleting versions, and moving versions to another class history.

10.1 Schema Modification

Client Smalltalk and GemStone Smalltalk both have schema modification support. Client Smalltalk supports only a single instance of a class; when a class is modified, instance migration occurs immediately. Because GemStone stores persistent objects, schema modification is a more complex issue.

GemStone Smalltalk supports schema modification and protects the integrity of your stored data by allowing you to define different versions of classes. It keeps track of these versions in a class history object.

Every class in GemStone Smalltalk has a class history instance variable. A class history is an object that maintains a list of all versions of the class. Every GemStone class is listed in exactly one class history. You can define any number of different versions of a class and declare that the different versions belong to the same class history. You can also migrate some or all instances of one version of a class to another version when you need to. By default, migration of an object from one class version to another will preserve the values of unnamed instance variables and instance variables that have the same name in both classes.

It is not necessary for different versions of a class to have a similar structure or a similar implementation. The classes don't even need to have the same name, although it is probably less confusing if they do or if you establish and adhere to some naming convention.

The section entitled "Modifying an Existing Class" on page 63 explains how to create different versions of a class in GemBuilder.

Instance Migration Within GemStone

The migration operation in GemStone is flexible and configurable.

- Instances of any class can migrate to any other, as long as they share a class history. The two classes need not be similarly named or have anything else in common.
- Migration can occur whenever you want it to.
- You don't have to migrate all instances of a class at once; you can migrate only certain instances as needed.

- You can choose which values of the old instance variables are used to initialize values of the new instance variables. overriding the default mapping mechanism as necessary.

Setting the Migration Destination

You can use the message `migrateTo:` to set a migration destination in the class that you need to migrate from as follows:

OldClass `migrateTo:` *NewClass*

This message merely lets the class know its migration destination; it does not cause migration to occur. Migration takes place only when the class receives one of the `migrateInstances` messages described in the section "Migrating Objects."

It is not necessary to set a migration destination ahead of time; you can specify the destination class when you decide to migrate instances. It is also possible to set a migration destination and then migrate the instances of the old class to a completely different class by specifying a different migration destination as part of the message that performs the migration.

You can erase the migration destination for a class by sending it the message `cancelMigration`, and you can query the migration destination by sending `migrationDestination` to the class.

Migrating Objects

A number of mechanisms are available to allow you to migrate one instance or a specified set of instances to a previously specified migration destination or to another explicitly specified destination.

You can execute the following expression to identify instances that may need to be migrated:

`SystemRepository listInstances:` *anArrayOfClasses*`.`

The `listInstances:` message takes as its argument an array of classes and returns an array of sets. The contents of each set consists of all instances whose class is equal to the corresponding element in the argument *anArrayOfClasses*. Instances to which you lack read authorization are omitted without notification.

The simplest way to migrate an instance of an older class is to send the message `migrate` to the instance. If the object is an instance of a class for which a migration destination has already been defined, the object becomes an instance of the specified version of the class. If no destination has been defined, no change occurs.

You can bypass the migration destination or migrate instances of classes for which no migration destination has been specified by specifying the destination directly in the message that performs the migration.

The following messages (defined in class Class) specify a one-time-only operation that ignores any preset migration destination class.

`migrateInstances:`*aCollectionOfInstances* `to:`*DestinationClass*

`migrateInstancesTo:`*DestinationClass*

The `migrateInstances:to:` message migrates specified instances to a class; the `migrateInstancesTo:` migrates all instances of the receiver to a class.

Things to Watch Out For

There are a few things that you should be aware of when migrating objects.

- You cannot send a `migrate` message to `self`. Attempting to do so generates an error that reports "The object you are trying to migrate was already on the stack."
- You cannot migrate instances that you are not authorized to read or write.
- You need to be aware that the instance variable map used in migrating instances from one GemStone class to another is not the same as the instance variable map described in Chapter 5, whose purpose is to map instance variables from GemStone to Smalltalk.

Instance Variable Mapping in Migration

GemStone supports instance migration between two classes that belong to the same class history. For simple migrations, such as the addition or removal of an instance variable, GemStone provides a default migration mechanism that copies data from each instance variable of the old object to the instance variable of the same name in the new object (if one exists). You can write methods to customize this migration on a class-by-class basis.

When an object is migrated, it refers to the class and class instance variables that have been defined for the new version of the class. These variables have whatever values have been assigned to them in the class object.

The simplest way to retain the data held in instance variables is to use instance variables having the same names. If two versions of a class have instance variables with the same name, the values of those variables are automatically retained when the instances migrate from one class to the other.

However, the structure of the two classes may be different, and a one-to-one mapping may not be possible. For example, if the new class has an instance variable for which no corresponding variable exists in the old class, that instance variable is initialized to *nil* upon migration. Similarly, if the old class has an instance variable for which no corresponding variable exists in the new class, the value of the old variable is dropped and the data it represents is no longer accessible from that object.

You may encounter situations in which you want to initialize a variable having one name with the value of a variable having a different name. This requires providing an explicit mapping from the instance variable names of the older class to the instance variable names of the migration destination. To do this you will need to override the default mapping strategy by reimplementing a class method named `instVarMappingTo:` in your destination class. This method is defined in Class to return an instance variable mapping from the receiver's named instance variables to those in the other class, but it can be customized in the new class to explicitly map the two different names.

There also may be times when you need to perform a specific operation on the value of a given variable before initializing the corresponding variable in the class to which the object is migrating.

For example, suppose that you have a class named Point, which defines two instance variables: *x* and *y*. These instance variables define the position of the point in Cartesian two-dimensional coordinate space. Now suppose that you define a class named NewPoint to use polar coordinates. The class has two instance variables named *radius* and *angle*. The default mapping strategy would cause Point objects to completely lose their position because the old and new classes have no instance variables in common.

This can be handled, however, by overriding a migration method in NewPoint by defining it to include an operation that transforms the values of *x* and *y* into values that can properly be assigned to *radius* and *angle*. In this case, the appropriate method to override is `migrateFrom:instVarMap:`. Then, when you migrate an instance of Point to an instance of NewPoint, the migration code that calls `migrateFrom:instVarMap:` executes the method in NewPoint instead of the one in Object that defines the default behavior. (This example is explained in detail in the *GemStone Programming Guide*.)

10.2 Schema Coordination

GemBuilder's goal in supporting schema migration is to provide an interaction between the client Smalltalk and the GemStone server that provides as much of GemStone's capabilities as possible, while minimizing the impact on the client Smalltalk system.

GemBuilder preserves the behavior of having only a single version of a given class in client Smalltalk at one time. That client Smalltalk class will be mapped to a specific version of a GemStone server class, resolved at login time (if a connector is defined) by its name. If, while faulting an object into client Smalltalk, GemBuilder discovers that the server object is an instance of a class that is a different version of the class that is in client Smalltalk, and the class version of the server object is less than the class version that is already mapped to the client class, it will be faulted in in the format of the class in client Smalltalk and flagged so that if it is modified and written back to the server, the server instance will be of the newer class version. This will lazily-migrate forward server instances to newer class versions.

For example, suppose you have a class named `Foo` on the GemStone server, and there are two versions of it: `Foo [1]` and `Foo [2]`. Suppose that client Smalltalk has a representation of `Foo [2]`. Instances of `Foo [2]` are replicated back and forth between client and server, as usual. If GemBuilder attempts to fault an instance of `Foo [1]`, however, GemBuilder will discover that there is no class mapping for `Foo [1]`. GemBuilder will then do the following:

1. It will fetch the name of the GemStone server class and discover that there is a client Smalltalk class by the same name that is already mapped to a server class.
2. It will verify that the two server classes are in the same class history, and that the version of `Foo [1]` is less than the version of `Foo [2]`.
3. It will then ask GemStone to make a copy of the `Foo [1]` instance, migrate the copy to `Foo [2]`, and replicate that migrated copy to the client. The client replicate is mapped to the original `Foo [1]` server instance.
4. If the client replicate is later modified in client Smalltalk, marked dirty, and flushed to the server, it will not be migrated back, and the server object will become an instance of `Foo [2]`.

This process is fairly expensive. If you are running GemBuilder in verbose mode, the discovery of an client Smalltalk class that is mapped to an old version of a GemStone server class (a version that is not the migration destination) will be logged to the transcript. If you see this happening frequently, you should consider

migrating your instances to the GemStone server class version corresponding to your client Smalltalk class.

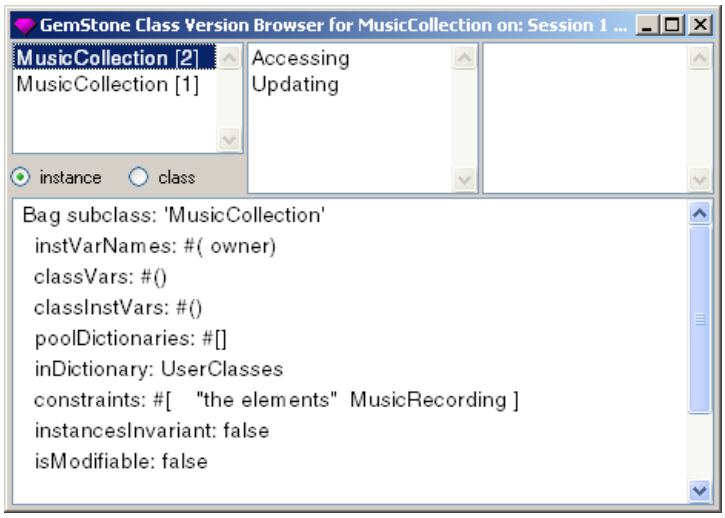
10.3 The Class Version Browser

The Class Version Browser is a specialized Class Browser that can be used for examining a class history, inspecting instances, migrating instances, deleting versions, and moving versions to another class history.

To open a Class Version Browser, select a class in a browser and choose **Browse Versions** from the Classes menu. If more than one version of a class has been created, the class list in the spawned browser displays the version number next to the class name.

A Class Version Browser is shown in Figure 10.1.

Figure 10.1 The Class Version Browser



Menus in the Class Version Browser

For the most part, the Class Version Browser's menus are the same as the menus in the Class Browser. However, the Class Version Browser's Classes menu contains the additional items **inspect instances** and **migrate instances...**. Also,

note that the Classes menu items **Move...** and **Remove...** behave differently in this browser.

The layout of the browser is similar to the Class Browser. The Method Category and Message menus are the same as in a spawned Class Browser. The Classes menu, however, has additional functionality.

The commands available in the Class Version Browser are shown in Table 10.1:

Table 10.1 Class Menu in Class Version Browser

file out as...	Writes GemStone Smalltalk code defining the selected class version and all of its methods to be written to a file in Topaz format. The class and its methods can later be re-created (read from the file and recompiled) by means of a command given from the File List Browser. See "Saving Class and Method Definitions in Files" on page 65.
file out methods as...	Writes GemStone Smalltalk code defining the selected class version's methods to be written to a file in Topaz format. See "Saving Class and Method Definitions in Files" on page 65 for more information on filing out.
browse class	Opens a Class Browser that includes only the selected class version.
browse hierarchy	Opens a Class hierarchy Browser that includes superclasses and subclasses of the selected class version.
browse versions	Opens another Class Version Browser on this class history.
browse references	This menu item has two submenus: **to this version of this class** and **to any version of this class**. Opens a method list browser containing all methods whose compiled code contains a reference to this version of the class, or to any version of the class in this class history.
hierarchy	Lists the superclasses and subclasses of the current class. Any instance variable names declared in a class appear in the hierarchy list in parentheses.
definition	Displays the definition (that is, the subclass creation message) of the currently selected GemStone class version. This is shown by default.

Table 10.1 Class Menu in Class Version Browser(Continued)

comment	Displays the class comment, if one exists.
move to...	Moves the selected class version to another class history. Prompt for a target class, adds the selected version to the target class's class history, and updates the browser. The class name of the selected version is changed to that of the target class.
remove...	Remove the selected class version from the class history. Upon confirmation to proceed, asks if the user wants to migrate instances. If yes, prompts for the migration target, migrates the instances and updates the browser.
create access	Creates methods for accessing and updating the instance variables of the currently selected class version.
create in ST	Generate the selected class in client Smalltalk, if a mapping doesn't already exist. If it does exist, executing this menu item has no effect.
compile in ST	Attempts to compile all methods (instance and class) of selected class version in corresponding client Smalltalk class.
update	Update the browser to the current view of the class versions.
inspect instances	Open an inspector on instances on the selected version.
migrate instances...	Migrate all instances of the selected versions. Prompts you to select which version to migrate to. The user can only migrate to another version of the same class history, so if all versions are selected there is no migration destination and the item should be grayed out. Otherwise, prompt for the version to migrate to by popping up a list of versions not selected. Allow the user to cancel the operation by clicking a cancel button.

Chapter

11 Performance Tuning

This chapter discusses ways that you can tune your GemBuilder application to optimize performance and minimize maintenance overhead.

Profiling
explains ways you can examine your program's execution.

Selecting the Locus of Control
provides some rules of thumb for deciding when to have methods execute on the client and when to have them execute on the server.

Replication Tuning
explains the replication mechanism and how you can control the level of replication to optimize performance

Optimizing Space Management
explains how you can reclaim space from unneeded replicates.

Using Primitives
introduces the use of methods written in lower-level languages such as C.

Multiprocess Applications
discusses nonblocking protocol and process-safe transparency caches.

See Appendix A, "GemBuilder Configuration Parameters", for GemBuilder configuration parameters that can used to tune performance.

For further information, see the *GemStone Programming Guide* for a discussion on how to optimize GemStone Smalltalk code for faster performance. That manual explains how to cluster objects for fast retrieval, how to profile your code to determine where to optimize, and discusses optimal cache sizes to improve performance.

11.1 Profiling

Before you can optimize the performance of your application, you must find out where most of the execution time is being spent. There are client Smalltalk tools available for profiling client code. GemStone also has a profiling tool in the class ProfMonitor. This class allows you to sample the methods that are executed in a given block of code and to estimate the percentage of total execution time represented by each method, within GemStone server execution. See the chapter on performance in the *GemStone Programming Guide* for details.

Profiling Client Smalltalk Execution

GemBuilder can be configured to collect statistics describing the performance of its internal operation. These statistics are archived to a file (a statistics archive file), which can be viewed by a tool called VSD (for more information on VSD, See “VSD” on page 203). Statistics tracking introduces minimal overhead into GBS. A VisualWorks process named “GBS Stat Monitor” samples and archives the stats at a regular, configurable time interval.

To enable the tracking of all GBS statistics in an image, and start the statistics monitor archiving at a default interval of once every 2000 milliseconds, execute the following:

```
GBSM statsEnabled: true
```

To disable all statistics tracking and turn off the statistics monitor, execute:

```
GBSM statsEnabled: false
```

To check if any statistics are currently enabled:

```
GBSM statsEnabled
```

Main Statistics

GBS provides two groups of statistics, called “main statistics” and “cache inventory statistics”. The “main statistics” of GBS are statistics associated with the

session manager and with each logged in session. The session manager statistics are:

Table 11.1 Session Manager Main Statistics

Statistic	Description
numSessions	The number of logged in GbsSessions
sessionListProtectInvocations	The number of accesses to the sessionListProtect semaphore
gciCallProtectInvocations	The number of accesses to the GciCallProtect semaphore (a GbxCInterface shared variable)
clientMapSize	The number of entries in the client map (stObjectCache)
mainStatSampleTime	The amount of time spent sampling manager and session main statistics
cacheStatSampleTime	The amount of time spent sampling cache statistics

The statistics associated with each logged in session are:

Table 11.2 Session Main Statistics

Statistic	Description
gciCallsToGem	The number of gci calls made that communicate with the gem
gciCallsToGemTime	The amount of time spent in gci calls that communicate with the gem
traverseCalls	The number of traversal calls (all types, including more traversal)
traverseCallTime	The amount of time spent in traversal calls
objectsTraversed	The total number of objects (with or without a value buffer) received by traversal call
bytesTraversed	The cumulative number of bytes returned by traversal calls

Table 11.2 Session Main Statistics (Continued)

Statistic	Description
traversalUnpackingTime	The total number of milliseconds spent unpacking traversal buffers
storeTraversals	The number of store traversal calls made
bytesSentByStoreTraversal	The number of bytes sent cumulatively by store traversal calls
objectsStoredByTraversal	The total number of objects stored by store traversal calls
sigAborts	The number of signaled aborts received
lostOtRoots	The number of lostOTRoot signals received
changedObjNotifications	The number of changed object notifications received
sessionSignals	The number of gem-to-gem signals received
freeOopsFetched	The number of free oops fetched
sessionProtectInvocations	The number of times the sessionProtect semaphore has been invoked
gciErrors	The number of errors reported by GciErr()
nbEndResultReady	The number of times GciNbEnd() was called and a result was ready
nbEndResultProgressed	The number of times GciNbEnd() was called and progress was made
nbEndResultNoProgress	The number of times GciNbEnd() was called when the result wasn't ready and no progress was made
serverMapSize	The number of entries in this session's server map (gsObjectCache)

To enable the main statistics without enabling cache statistics, execute:

```
GBSM mainStatsEnabled: true
```

Unlike the all-in-one “statsEnabled:” method, this method doesn't start the statistics monitor. To start the statistics monitor, execute:

```
GBSM statMonitorRunning: true
```

To specify a specific statistics archiving interval in milliseconds, execute:

```
GBSM statSampleInterval: milliseconds
```

Cache Inventory Statistics

GBS provides cache inventory statistics, which show the number of instances of, and bytes consumed by, each class of object found in the clientMap (formerly the stObjectCache). To enable cache inventory statistics without enabling main statistics, execute:

```
GBSM cacheStatsEnabled: true
GBSM statMonitorRunning: true
```

Cache inventory statistics are more expensive to sample and archive than the main GBS statistics. Because of this, cache statistics are not sampled and archived every time the statistics monitor performs sampling and archiving of the main statistics. By default, the cache statistics are sampled and archived every 5th time. This value is configurable by sending:

```
GBSM cacheSampleIntervalMultiplier: anInteger
```

This value times statSampleInterval is the interval between two cache statistics samples. For example, with the default cacheSampleIntervalMultiplier of 5 and the default statSampleInterval of 2000 milliseconds, the cache statistics will be sampled and archived every 10000 milliseconds (or once every 10 seconds). A cacheSampleIntervalMultiplier of 1 would mean that cache statistics will be sampled and archived every time the main statistics are sampled.

VSD

The Visual Statistics Display tool, VSD, is provided with the GemStone server product, and can be found in the directory $GEMSTONE/bin/vsd. However, the version of VSD that is shipped with earlier versions of the GemStone/S 6.x server product is incompatible with the GBS statistics archive format. The version of VSD that is shipped with the GemStone/S 64 Bit server can be used to read GBS statistics files.

You can download the latest version of VSD from:

http://support.gemstone.com/gemstone_s/downloads/add_ons/vsd4/index.htm

To view one or more GBS statistics files, invoke vsd with the statistic files as arguments. See the help system in VSD for more information.

11.2 Selecting the Locus of Control

By default, GemBuilder executes code in the client Smalltalk. Objects are stored in GemStone for persistence and sharing but are replicated in the client Smalltalk for manipulation. In general, this policy works well. There are times, however, when it is preferable or required to execute in GemStone.

One motivation for preferring execution in GemStone is to improve performance. Certain functions can be performed much more efficiently in GemStone. The following section discusses the trade-offs between client Smalltalk and server Smalltalk execution and how to choose one space over the other.

Beyond optimization, some functions can be performed *only* in GemStone. GemStone's System class, for example, cannot be replicated in the client Smalltalk; messages to System have to be sent in GemStone.

Locus of Execution

This section centers on controlling the locus of execution—in other words, determining whether certain parts of an application should execute in the client Smalltalk or in GemStone. Subsequent sections discuss other ways of tuning to increase execution speed.

Client Smalltalk and GemStone Smalltalk are very similar languages. Using GemBuilder, it is easy to define behavior in either client Smalltalk or GemStone to accomplish the same task. There are, however, performance implications in the placement of the execution. This section discusses several factors to weigh when choosing the space in which to execute methods.

Relative Platform Speeds

One consideration when choosing the execution platform is the relative speed of the client Smalltalk and the server Smalltalk execution environments. Your client Smalltalk will often run faster than GemStone on the same machine. GemStone's database management functions and its ability to handle very large data sets add some overhead that the client Smalltalk environment doesn't have.

Cost of Data Management

Execution cannot complete until all objects required have been brought into the object space. When executing in the client Smalltalk, this means that all GemStone server objects required by the message must be faulted from GemStone. When executing in GemStone, this means that dirty replicates must

be flushed from the client Smalltalk. In general, it is impossible to tell exactly which objects will be required by a message send, so GemBuilder flushes all dirty replicates *before* a GemStone message send and faults all dirty GemStone server objects *after* the send.

Clearly, data movement can be expensive. Although the client Smalltalk environment might be more efficient for some messages, faulting the object into the client Smalltalk might overwhelm the savings. If the objects are all already there, however, or if the objects will be reused for other messages, then the movement may be justified.

For example, consider searching a set of employees for a specific employee, giving her a raise, and then moving on to another unrelated operation. Although a brute force search may be faster in your client Smalltalk, the cost of moving the data to the client may exceed the savings. The search should probably be done in GemStone.

However, if additional operations are going to be done on the employee set, the cost of moving data is amortized and, as the number of operations increases, becomes less than the potential savings.

GemStone Optimization

Some optimizations are possible only using GemStone server execution. In particular, repository searching and sorting can be done much more quickly on the GemStone server than in your client Smalltalk as data sets become large.

If you will be doing frequent searches of data sets such as the employee set in the previous example, using an index on the server Smalltalk set will speed execution.

The *GemStone Programming Guide* provides a complete discussion of indexes and optimized queries.

11.3 Replication Tuning

The faulting of GemStone server objects into the client Smalltalk is described in Chapter 5. As described there, a GemStone server object has a replicate in the client Smalltalk created for itself, and, recursively, for objects it contains to a certain level, at which point stubs instead of replicates are created.

Faulting objects to the proper number of levels can noticeably improve performance. Clearly, there is a cost for faulting objects into the client Smalltalk. This is made up of communication cost with GemStone, object

accessing in GemStone, object creation and initialization in the client Smalltalk, and increased virtual machine requirements in the client Smalltalk as the number of objects grows. For this reason, you should try to minimize faulting and fault in to the client only those objects that will actually be used in the client.

On the other hand, inadequate faulting also has its penalties. In the RPC version of GemBuilder, communication overhead is important. When fetching an employee object, it is wasteful to stub the name and then immediately fetch the name from GemStone. Even in the linked version, it is better to avoid creating the stub and then invoking the fault mechanism when sending it a message.

Controlling the Fault Level

By default, two levels of objects are faulted with the linked version of GemBuilder, and four levels are faulted for the RPC version. This reflects the cost of remote procedure calls and the judgment that it is better to risk fetching unneeded objects to avoid extra calls to GemStone.

It is possible to tune the levels of stubbing to a more optimal level with a knowledge of the application being programmed. You can set the configuration parameters `faultLevelRpc` and `faultLevelLnk` to a SmallInteger indicating the number of levels to replicate when updating an object from GemStone to the client Smalltalk. A level of 2 means to replicate the object and each object it references, stubbing objects beyond that level. A level of 0 indicates no limit; that is, entering 0 prevents any stubs from being created. The default for the linked version is 2; the default for the RPC version is 4. To examine or change this parameter, choose **GemStone > Browse > Settings** and select the **Replication** tab in the resulting Settings Browser.

NOTE

Take care when using a level of 0 to control replication. GemStone can store more objects than can be replicated in a client Smalltalk object space.

Preventing Transient Stubs

If the default `faultLevelLnk` or `faultLevelRpc` is the only mechanism used to control fault levels, it is possible to create large numbers of stubs that are immediately unstubbed.

To prevent stubbing on a class basis, reimplement the `replicationSpec` class method for that class. For details, see “Replication Specifications” on page 99.

Setting the Traversal Buffer Size

The traversal buffer is an internal buffer that GemBuilder uses when retrieving objects from GemStone. The larger the traversal buffer size, the more information GemBuilder is able to transfer in a single network call to GemStone. To change its value, send the message

```
GbsConfiguration current traversalBufferSize: aSmallInteger.
```

You can also change this value by using the Settings Browser: choose **GemStone > Tools > Settings** and select the **Server Communications** category in the resulting Settings Browser.

11.4 Optimizing Space Management

In normal use of GemBuilder, objects are faulted from GemStone to the client Smalltalk on demand. In many ways, however, this is a one-way street, and the client Smalltalk object space can only grow. Advantages can be gained if client Smalltalk replicates can be discarded when they are no longer needed. A reduced number of objects on the client reduces the load on the virtual machine, garbage collection, and various other functions.

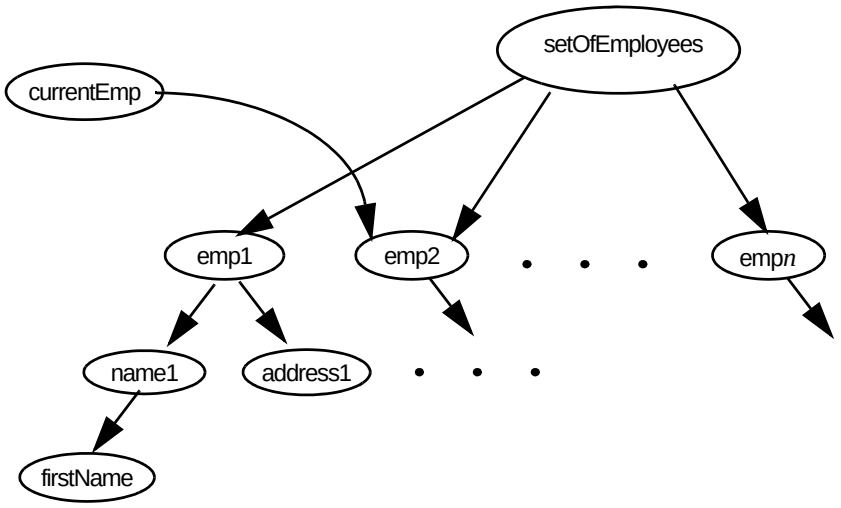
Measures you can take to control the size of the client Smalltalk object cache include explicit stubbing, using forwarders, and not caching certain objects.

Explicit Stubbing

If the application knows that a replicate is not going to be used for a period of time, the space taken by that object can be reclaimed by sending it the message `stubYourself`. More importantly, any objects it references become candidates for garbage collection in your client Smalltalk.

Consider having replicated a set of employees. After faulting in the set and the objects transitively referenced from that set, the objects in the client Smalltalk look something like this.

Figure 11.1 Employee Set Faulted into the Client Smalltalk



Clearly, there can be a large number of objects referenced transitively from the employee set. If the application's focus of interest changes from the set to, say, a specific employee, it may make sense to free the object space used by the employee set.

In this example, one solution is to send `stubYourself` to the `setOfEmployees`. All employees, except those referenced separately from the set, become candidates for garbage collection.

Of course, if the application will be referencing the `setOfEmployees` again in the near future, the advantage gained by stubbing could be offset by the increased cost of faulting later on.

Also, be aware of the difference between two ways of modifying the value of an instance variable: by using an access method and by direct assignment. For example, consider an object with an instance variable named instVarX. You can assign the value 5 to instVarX in two ways:

```
insVarX := 5          (direct assignment)
self instVarX: 5      (access method)
```

When the object is replicated in your Smalltalk workspace, each of these assignments yields the same result. When the object is represented in the Smalltalk workspace by a stub, however, the stub must be faulted in as a replicate ("unstubbed") before the assignment can occur. The access method causes the

stub to be faulted in and yields the correct result. Direct assignment, however, does not cause the stub to be faulted in and can cause errors:

```
self stubYourself.
self instVarX: 5.          (reliable)

self stubYourself.
instVarX := 5.             (unreliable)
```

Using Forwarders

Another solution is to declare the `setOfEmployees` as a forwarder. See "Forwarders" on page 86.

11.5 Using Primitives

Sometimes there is an advantage to dropping out of Smalltalk programming and writing methods in a lower-level language such as C. Such methods are called *primitives* in Smalltalk; GemStone refers to them as *user actions*. There are serious concerns when doing this. In general, such applications will be less portable and less maintainable. However, when used judiciously, there can be significant performance benefits.

In general, profile your code and find those methods that are heavily used to be candidates for primitives or user actions. The trick to proper use of primitives or user actions is to create as few as possible. Excess primitives or user actions make the system more difficult to understand and place a heavy burden on the maintainer.

For a description about adding primitives to your client Smalltalk, see the vendor's documentation. For adding user actions to GemStone, see the *GemBuilder for C* user manual.

11.6 Multiprocess Applications

Some applications support multiple Smalltalk processes running concurrently in a single image. In addition, some applications enter into a multiprocess state occasionally when they make use of signalling and notification. Multiprocess GemBuilder applications must exercise some precautions in order to preserve expected behavior and data integrity among their concurrent processes.

Blocking and Nonblocking Protocol

In a linked GemBuilder session, GemStone operations execute synchronously: the application must wait for a GemStone operation to complete before proceeding with the execution process that called it. Synchronous operation is known in GemBuilder as *blocking protocol*.

An RPC GemBuilder session can support asynchronous operation: *nonblocking protocol*. When the configuration parameter **blockingProtocolRpc** is `false` (the default setting in RPC sessions), client Smalltalk processes can proceed with execution during GemStone operations. A session, however, is permitted only one outstanding GemStone operation at a time.

When **blockingProtocolRpc** is `true`, behavior is the same as in a linked session: the execution process must wait for a GemStone call to return before proceeding.

One Process per Session

Applications that limit themselves to one process per GemStone session are relatively easy to design because each process has its own view of the repository. Each process can rely on GemStone to coordinate its modifications to shared objects with modifications performed by other processes, each of which has its own session and own view of the repository. For such applications, setting **processSafeCaches** to true is the only additional precaution required. If at all possible, try to limit your application to one process per GemStone session.

Multiple Processes per Session

Applications that have multiple processes running against a single GemStone session must take additional precautions.

You may not have designed your application to run multiple processes under a single GemStone session. However, if your application uses signals and notifiers, chances are it is occasionally running two processes against a single GemStone session. Methods that create concurrent processes include:

```
GbsSession
   >>notificationAction:
   >>gemSignalAction:
   >>signaledAbortAction:
```

When the specified event occurs, the block you supply to these methods runs in a separate process. Unless your main execution process is idle when these events

occur, you need to take the same precautions as any other application running multiple processes against a single session.

Applications that have multiple processes running against a single GemStone session should take these additional precautions:

- coordinate transaction boundaries
- coordinate flushing
- coordinate faulting

GemBuilder provides a method, `GbsSession>>critical:` *aBlock*, that evaluates the supplied block under the protection of a semaphore that is unique to that session. The best approach to creating an application that must support more than one process interacting with a single GemStone session is to organize its logical transactions into short operations that can be performed entirely within the protection of `GbsSession>>critical:`. All of that session's commits, aborts, executes, forwarder sends, flushes and faults should be performed within `GbsSession>>critical:` blocks.

For example, a block that implements a writing transaction will typically start with an abort, make object modifications, and then finish with a commit. A block that implements a reading transaction might start with an abort, perhaps perform a GemStone query, and then maybe display the result in the user interface.

Coordinating Transaction Boundaries

Multiple processes need to be in agreement before a commit or abort occurs. For example, suppose two processes share a single GemStone session. If one process is in the process of modifying a set of persistent objects and a second process performs a commit, the committed state of the repository will contain a logically inconsistent state of that set of objects.

The application must coordinate transaction boundaries. One way to do this is to make one process the transaction controller for a session, and require that all other processes sharing that session request that process for a transaction state change. The controller process can then be blocked from performing that change until all other processes using that session have relinquished control by means of some semaphore protocol.

Coordinating Flushing

GemBuilder's transparency mechanism flushes dirty objects to GemStone whenever a commit, abort, GemStone execution or forwarder send occurs. Whenever a process modifies persistent objects, it must protect against other

processes performing operations that trigger flushing of dirty objects to GemStone. The risks are that a flush may catch a logically inconsistent state of a single object, or might cause GemBuilder to mark an object "not dirty" without really flushing it.

To control when flushing occurs, perform update operations within a block passed to `GbsSession>>critical:`.

Coordinating Faulting

If two processes send a message to a stub at roughly the same time, one of the processes can receive an incomplete view of the contents of the object. This results in doesNotUnderstand errors which cannot be explained by looking at them under a debugger, because by the time it is visible in the debugger, the object has been completely initialized. Unstubbing conflicts can be avoided by encapsulating potential unstubbing operations within the protection of a `GbsSession>>critical:` block.

Using the VisualWorks Application Model

As mentioned above, an application that otherwise restricts itself to one process per GemStone session can occasionally find itself running multiple processes against a single session through the asynchronous operation of signals and notifiers.

One way to coordinate the activities of signal and notifier processes is to remap asynchronous GemStone events into synchronous window events, as though they were received through the user interface. Implementing this strategy requires two steps. The first is to use some or all of the following methods to redirect GemStone signals and notifiers to the application model:

```
GbsSession
   >>sendGemSignalEventsTo: anApplicationModel
   >>sendNotificationEventsTo: anApplicationModel
   >>sendSignaledAbortEventsTo: anApplicationModel
```

For example, you could implement an initialize method in your application model as shown in Example 11.1.

Example 11.1

```
self session
sendNotificationEventsTo: self;
sendSignaledAbortEventsTo: self.
```

The second step is to reimplement `windowEvent:from:` in your ApplicationModel subclass to respond to the appropriate corresponding events `#gemSignalAction`, `#notificationAction`, or `#signaledAbortAction`, as shown in Example 11.2.

Example 11.2

```
windowEvent: anEvent from: aWindow
    super windowEvent: anEvent from: aWindow.
    (anEvent key == #signaledAbortAction) ifTrue: [
    Transcript cr;
show: 'Aborting and refreshing the view.'.
self session abortTransaction.
self updateView ].
    (anEvent key == #notificationAction) ifTrue: [
    (anEvent value includes: someList) ifTrue: [
Transcript cr;
    show: 'List modified by another session';
    cr; show: 'aborting and refreshing view.'.
self session abortTransaction.
self updateView ]]
```

Appendix

A GemBuilder Configuration Parameters

GemBuilder provides configuration settings that allow GemBuilder to operate differently for development or debugging, control details of the user interface, and tune your program for performance.

This appendix describes the Settings Browser and GemBuilder configuration parameters, their default and legal values, and their significance.

A.1 Setting Configuration Parameters

Some configuration parameters control fundamental features of GemBuilder, and must remain the same while the image is running. Other parameters can be modified while GemBuilder is running, and may take effect immediately or at a later point, or can be set individually for a session to override the global behavior. The configuration parameter descriptions starting on page page 217 provide specific details for each parameter.

Global settings

When sessions log in, they obtain an initial set of configuration parameters based on the configuration settings in the current global GbsConfiguration.

To determine the current global setting of a parameter, send the parameter name as a message to the global instance of GbsConfiguration, GbsConfiguration current. For example, the following expression returns the setting of the **connectVerification** parameter:

```
GbsConfiguration current connectVerification
false
```

To globally set a parameter, append a colon to the parameter name and send it as a message to the GbsConfiguration instance, with the desired value as the argument. For example, to set the **connectVerification** parameter, send:

```
GbsConfiguration current connectVerification: true
```

You may also use the Settings Browser to view and change the settings of these parameters. (See “The Settings Browser” on page 231.)

Session-specific settings

While many configuration parameters apply to the image as a whole, other parameters may be modified for specific sessions.

For these parameters, the value in GbsConfiguration current is used at login. Subsequently, you may send the GbsConfiguration messages to the the session's configuration (acquired by sending #configuration to the session) to determine or modify the value for that session only.

For example, if the current session requires a larger traversal buffer, an expression such as the following will increase the size for this session, while leaving the global setting for new sessions unchanged.

```
GBSM currentSession configuration traversalBufferSize:
500000.
```

A.2 GemBuilder Configuration Parameter

The following table summarizes GemBuilder configuration parameters. Each parameter is described in detail following the table.

Table A.1 Configuration Parameters for GemBuilder

Parameter	Legal values	Default	Scope
alwaysUseGemCursor	true/false	true	Global
assertionChecks	true/false	false	Global
autoMarkDirty	true/false	true	Global
blockingProtocolRpc	true/false	false	Global
blockReplicationEnabled	true/false	true	Global
blockReplicationPolicy	#replicate/ #callback	#replicate	Global
bulkLoad	true/false	false	Global
clientMapCapacity	any integer	30000	Global
clientMapFinalizerPriority	integers between 1 and 99, inclusive	30	Global
confirm	true/false	true	Global
connectorNilling	true/false	true	Session-specific
connectVerification	true/false	false	Global
defaultFaultPolicy	#immediate/#lazy	#lazy	Global
deprecationWarnings	true/false	true	Global
eventPollingFrequency	any integer	300	Global
eventPriority	any integer	50	Session-specific
faultLevelLnk	any integer	2	Session-specific
faultLevelRpc	any integer	4	Session-specific
forwarderDebugging	true/false	false	Global
freeSlotsOnStubbing	true/false	true	Global
fullCompression	true/false	false	Global

Table A.1 Configuration Parameters for GemBuilder (Continued)

Parameter	Legal values	Default	Scope
gcedObjBufferSize	any integer	2000	Global
generateClassConnectors	true/false	true	Session-specific
generateClientClasses	true/false	true	Session-specific
generateServerClasses	true/false	true	Session-specific
InitialDirtyPoolSize	any integer	100	Session-specific
libraryName	any string	empty string	Global
pollForAsynchronousEvents	true/false	false	Global
pollForRpcResponse	true/false	false	Global
removeInvalidConnectors	true/false	false	Global
rpcSocketWaitTimeoutMs	any integer	100	Session-specific
serverMapLeafCapacity	any integer	400	Session-specific
stubDebugging	true/false	false	Global
traversalBufferSize	any integer	250000	Session-specific
verbose	true/false	true	Global

alwaysUseGemCursor

Used to reduce the number of conditions under which GBS switches cursors. When true, GBS changes the cursor to the “gem” cursor during all interactions with the server. When false, the cursor is only changed during some server operations by the GBS tools.

Legal values: true / false
Default: true
Settings Tool tab: User Interface
Scope: Global

assertionChecks

This parameter is for the use of GemStone customer support.

Legal values: true / false
Default: false
Settings Tool tab: Debugging
Scope: Global

autoMarkDirty

Defines whether modifications to client objects are automatically detected. When false, the application must explicitly send markDirty to a client object after it has been modified, so GemBuilder will know to update the object in GemStone. Do not change this setting while sessions are logged in from this client process.

Legal values: true / false
Default: true
Settings Tool tab: Replication
Scope: Global

blockingProtocolRpc

Determines whether to use blocking or nonblocking protocol for RPC sessions (linked sessions cannot be non-blocking). When `false`, nonblocking protocol is used, enabling other threads to execute in the image while one or more threads are waiting for a GemStone call to complete. When `true`, GemBuilder must wait for a GemStone call to complete before proceeding with the thread that called it. Should not be changed for sessions that are already logged in.

Legal values: true / false
Default: false
Settings Tool tab: Server Communication
Scope: Session-specific

blockReplicationEnabled

When false, GemBuilder raises an exception when block replication is attempted—useful in determining if your application depends on block replication.

Legal values: true / false
Default: true
Settings Tool tab: Replication
Scope: Global

blockReplicationPolicy

Block replication requires decompiling and compiling the source code for blocks at runtime. Since it is usually not possible to include the Smalltalk compiler in a runtime image, block replication may cause problems in runtime applications. Block callbacks use client forwarders to evaluate the block in the client. Block callbacks escape the documented limitations of block replication, but do not perform well for blocks invoked repeatedly from GemStone.

Legal values: `#replicate` or `#callback`
Default: `#replicate`
Settings Tool tab: Replication
Scope: Global

bulkLoad

This parameter has no effect when logged into a GemStone / S 64 Bit server.

When `true`, newly created objects are stored in GemStone as permanent objects immediately, bypassing a step wherein they are temporary and eligible for storage reclamation by the GemStone garbage collector unless other objects refer to them (in which case they become permanent objects, as usual). Bypassing this step improves performance for bulk data loading. When `false`, the temporary object step is not bypassed.

Legal values: true / false
Default: false
Settings Tool tab: Cache Tuning
Scope: Global

clientMapCapacity

The minimum capacity in objects of the client object map. The client object map is used to map client replicates, stubs, and forwarders to their corresponding server objects. The map will grow in capacity if it runs out of room, and may shrink in

capacity if it has an excess of free space. The map will never shrink its capacity below **`clientMapCapacity`**. This value is also used as the initial capacity of the map, which is initialized only upon the first login after loading GBS into a clean image. Thus, changing **`clientMapCapacity`** will only affect the initial cache size if changed before the first login after GBS load. After that time, however, changing **`clientMapCapacity`** will limit the shrinkage of the map. Growing and shrinking the map take time, so performance-critical applications that replicate many objects may wish to have a larger map capacity. Check statistics for map size and grow / shrink events to see whether your map capacity is sufficient.

The legal value is any positive Integer; GBS will set the actual capacity to an appropriate value somewhat larger than specified.

Legal values: any positive Integer.
Default: 30000
Settings Tool tab: Cache Tuning
Scope: Global

clientMapFinalizerPriority

The process priority at which garbage-collected objects are finalized from the client object map. These must be finalized before the server can garbage-collect the corresponding server objects. By default, this finalization is done at a priority below the normal application priority. This allows finalization to run at times when the main application is waiting for user input or responses from the server. However, if your application seldom waits, and creates a lot of garbage replicates, it is possible that the finalizer might not get enough CPU cycles to keep up. If the unfinalized objects have no remaining references on the server, this will cause increased memory usage in the gem. If this is a problem, you may need to increase this setting to a priority above that of your application. Changing this value will change the finalizer's priority at the time of the next server interaction.

Legal values: an Integer between 1 and 99, inclusive
Default: 30
Settings Tool tab: Cache Tuning
Scope: Global

confirm

When `true`, you are prompted to confirm various GemBuilder actions. Leave set to `true` during application development; deployed applications may set to `false`.

Legal values: true / false
Default: true
Settings Tool tab: User Interface
Scope: Global

connectorNilling

When `true`, GemBuilder nils the Smalltalk object for certain session-based connectors after logout: all name, class variable, or class instance variable connectors whose postconnect action is **#updateST** or **#forwarder**. When the last session logs out, the Smalltalk object references of global connectors are also set to `nil`. Fast connectors, class connectors, and connectors whose postconnect action is **#updateGS** or **#none** are not set to `nil`. Clearing connectors that depend on being attached to GemStone server objects helps prevent defunct stub and forwarder errors.

When false, the logout sequence leaves the state of persistent objects in the image as it was.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send `#connectorNilling:` to the session's configuration to change the value for that session only. The session's current value will be used at logout.'

Legal values: true / false
Default: true
Settings Tool tab: Connectors
Scope: Session-specific

connectVerification

When `true`, connectors verify at login that they are not redefining a connector that already exists, and class connectors verify that the two classes they are connecting have compatible structures. When `false`, these things are not checked. Set to `true` during development unless logging in becomes too slow, or your connector definitions are stable. Applications in production should normally set this to false.

See “The Connector Browser” on page 136.

Legal values: true / false
Default: false
Settings Tool tab: Connectors
Scope: Global

defaultFaultPolicy

This parameter has no effect when logged into a GemStone / S 64 Bit server, which is always `#immediate`.

Specifies GemBuilder’s default approach to updating client Smalltalk objects whose GemStone counterparts have changed. When **`#lazy`**, GemBuilder responds to a change in a GemStone server object by turning its client Smalltalk replicate into a stub. The new GemStone value is faulted in the next time the stub is sent a message. When **`#immediate`**, GemBuilder responds to a change in a GemStone server object by updating the client Smalltalk replicate immediately. The defaultFaultPolicy is implemented by Object >> `faultPolicy`. Subclasses can override this method for specific cases.

Legal values: **`#immediate`** / **`#lazy`**
Default: **`#lazy`**
Settings Tool tab: Replication
Scope: Global

deprecationWarnings

When true, any attempt to use a deprecated feature of GemBuilder for Smalltalk causes a proceedable exception to be raised. Deprecated features may be removed in a future release. When false, deprecated features may be used with no warning.

Legal values: true / false
Default: true
Settings Tool tab: Debugging
Scope: Global

eventPollingFrequency

How often, in milliseconds, that GemBuilder polls for GemStone events such as changed object notification or Gem-to-Gem signaling.

Legal values: any Positive Integer
Default: 300
Settings Tool tab: Signals And Events
Scope: Global

eventPriority

The priority of the Smalltalk process that responds to GemStone events—that is, the priority at which the block will execute that was supplied as an argument to the keyword `gemSignalAction:, notificationAction:,` or `signaledAbortAction:`. These keywords occur in messages used by Gem-to-Gem signaling, changed object notification, or when GemStone signals you to abort so that it can reclaim storage, respectively.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #eventPriority: to the session's configuration to change the value for that session only. The priority will not change immediately, but the new value will be used the next time an action block is set and the event detection process is restarted.

Legal values: an Integer between 1and 99, inclusive
Default: 50
Settings Tool tab: Signals And Events
Scope: Session-specific

faultLevelLnk

The default number of levels to replicate an object from GemStone to client Smalltalk in a linked session.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #faultLevelLnk: to the session's configuration to change the value for that session only.

Legal values: any Integer
Default: 2
Settings Tool tab: Replication
Scope: Session-specific

faultLevelRpc

The default number of levels to replicate an object from GemStone to client Smalltalk in a remote session.

The value in GbsConfiguration current is used at login. Subsequently, you may send #faultLevelRpc: to the session's configuration to change the value for that session only.

Legal values: any Integer
Default: 4
Settings Tool tab: Replication
Scope: Session-specific

forwarderDebugging

When `true`, forwarders support debugging by responding to some basic messages locally, such as `printOn:, instVarAt:,` and `class`, which returns GbsForwarder. When `false`, these messages are forwarded to the GemStone server object.

Legal values: true / false
Default: false
Settings Tool tab: Debugging
Scope: Global

freeSlotsOnStubbing

When `true`, stubbing an existing replicate causes all persistent named instance variables (that is, those that will be faulted in when the stub is unstubbed) and all indexable instance variables to be set to `nil`, allowing stubs and their potentially outdated instance variables to be garbage collected if they become eligible. When false, GemBuilder does not alter instance variable values. To override this behavior on a class-by-class basis, reimplement `#freeSlotsOnStubbing` (inherited from Object).

Legal values: true / false
Default: true
Settings Tool tab: Replication
Scope: Global

fullCompression

When true, GemStone compresses all communication between the client and the server, reducing the amount of data sent across a network connection to an RPC gem. Has no effect on linked sessions. For network connections with low throughput, compression may improve overall performance. For fast enough network connections, compression may decrease overall performance due to the CPU time required to do compression and decompression.

This setting only takes effect at the time that a library is loaded (see **`libraryName`** below). If a library is loaded you will need to save your image, quit, and restart for a new **`fullCompression`** value to take effect.

Legal values: true / false
Default: false
Settings Tool tab: Server Communication
Scope: Global

gcedObjBufferSize

The initial size in objects of the buffer that holds server object IDs for objects which have been garbage collected in the client. The buffer is enlarged when necessary, but performance-sensitive applications that release many replicates at once may want to avoid this. The IDs in this buffer are sent to the server with each server interaction.

Legal values: any Integer
Default: 2000
Settings Tool tab: Cache Tuning
Scope: Global

generateClassConnectors

When `true`, a session connector is automatically created to connect two classes, one of which has been automatically generated in response to the presence of the other by the mechanisms described in the discussion of parameters **`generateClientClasses`** and **`generateServerClasses`**. When `false`, session connectors are not automatically created.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #generateClassConnectors: to the session's configuration to change the value for that session only.

See “Class Mapping” on page 83.

Legal values: true / false
Default: true
Settings Tool tab: Class Generation
Scope: Session-specific

generateClientClasses

When `true`, if a GemStone server object is fetched into the client Smalltalk image and the client Smalltalk image does not currently define the class of which it is an

instance, a corresponding class is defined in the image. When `false`, behavior is defined by the client Smalltalk image.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #generateClientClasses: to the session's configuration to change the value for that session only.

See “Class Mapping” on page 83.

Legal values: true / false
Default: true
Settings Tool tab: Class Generation
Scope: Session-specific

generateServerClasses

When `true`, if a client Smalltalk object is stored into GemStone and GemStone does not currently define the class of which it is an instance, a corresponding class is defined in GemStone Smalltalk. When `false`, GemBuilder raises an error.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #generateServerClasses: to the session's configuration to change the value for that session only.

See “Class Mapping” on page 83.

Legal values: true / false
Default: true
Settings Tool tab: Class Generation
Scope: Session-specific

InitialDirtyPoolSize

Initial size of the GbsSession dirtyPool identity set. For bulk loading, increasing this value reduces the number of times the set needs to grow. For applications that flush a small number of objects, decreasing this value (while keeping it larger than the number of objects being flushed) improves flushing performance.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send

#initialDirtyPoolSize: to the session's configuration to change the value for that session only. The new value will take effect after the next server operation.

Legal values: any positive Integer. GBS will select a prime size greater than this value.
Default: 100
Settings Tool tab: Cache Tuning
Scope: Session-specific

libraryName

The name of the DLL or shared library to use to contact the server. If this is set to an empty string, GBS loads the first found library with a default name. It tries loading linked libraries first (which support both linked and RPC logins), then RPC-only libraries. If a libraryName is specified, that exact library name is loaded. If the library is not found, an error is reported. On Unix or Linux, the library name may be specified as an absolute file path to the library file, or as a simple name (e.g. libgcirpc.so). On Windows, use a simple name. If a simple name is used, the library is found in the (platform-specific) standard directories for libraries. This setting does not affect any library that is already loaded. If a library is already loaded you will need to save your image, quit, and restart for a new libraryName to take effect.

Legal values: any String
Default: empty String
Settings Tool tab: Server Communication
Scope: Global

pollForAsynchronousEvents

This setting does not apply for linked sessions.

This setting determines which method to use to detect asynchronous events from the GemStone server. When `true`, GBS uses a timed polling loop, polling once every **`eventPollingFrequency`** milliseconds. When `false`, GBS waits for traffic on the socket that the gem uses for communication, and also polls once every **`eventPollingFrequency`** milliseconds. Under most circumstances, the socket wait provides more timely notification of these events.

Legal values: true / false
Default: false
Settings Tool tab: Signals And Events
Scope: Global

pollForRpcResponse

When using nonblocking protocol for RPC sessions, this setting determines which method to use to detect when a response has been received from the GemStone server. When `true`, GBS uses a timed polling loop. When `false`, it waits for traffic on the socket that the gem uses for communication. Under most circumstances, the socket wait provides better latency and uses slightly less CPU time.

This setting only affects non-blocking RPC sessions, and is ignored for linked sessions and for RPC sessions when **`blockingProtocolRpc`** is set to true.

Legal values: true / false
Default: false
Settings Tool tab: Server Communication
Scope: Global

removeInvalidConnectors

When `true` and **confirm** is `false`, if a connector fails to resolve at login, it is removed from the connector collections so that the issue does not arise again at next login.

When `true` and **confirm** is `true`, you are prompted to remove invalid connectors during login.

When `false`, invalid connectors are ignored.

See “The Connector Browser” on page 136.

Legal values: true / false
Default: false
Settings Tool tab: Connectors
Scope: Global

rpcSocketWaitTimeoutMs

When using the socket-wait style of detecting when an RPC response is ready (see the setting **`pollForRpcResponse`**), it is possible for the socket event to get lost occasionally, especially on Windows. To avoid waiting forever in that case, GBS double-checks for a response every **`rpcSocketWaitTimeoutMs`** milliseconds.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send

#rpcSocketWaitTimeoutMs: to the session's configuration to change the value for that session only.

Legal values: any positive Integer
Default: 100
Settings Tool tab: Server Communication
Scope: Session-specific

serverMapLeafCapacity

The lower bound in objects of the capacity of each leaf in the server map. The server map maps the IDs of server objects to their corresponding client replicates, forwarders, and stubs. The server map is structured as a shallow fixed-depth tree. Each node in the tree is identified by the upper bits of the object ID. Each leaf node is a hashed collection indexed by the lower bits of the object ID. The upper limit of the capacity for each leaf depends on the server product and version, ranging from 2^{22} to 2^{24} object IDs. Leaves are created on demand; only those leaves that actually contain objects exist. This setting controls the initial capacity of each leaf. Leaves will grow and shrink as necessary, but will not shrink below this setting. Growing and shrinking take some time, so performance-sensitive applications may want to adjust this value. Using a larger value decreases the time spent growing and shrinking each leaf, but increases memory use, and also increases the time spent initializing each leaf.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #serverMapLeafCapacity: to the session's configuration to change the value for that session only. From that point on, new leaf creation and all leaf shrinkage will be subject to the new value.

Legal values: any positive Integer. GBS will select a prime table size greater than this value, but not exceeding 2^{24}.
Default: 400
Settings Tool tab: Cache Tuning
Scope: Session Specific

stubDebugging

When `true`, stubs support debugging by responding to some basic messages locally, such as `printOn:, instVarAt:,` and `class`, which returns

GbxObjectStub. When `false`, these messages cause the stub to fault into the client image from GemStone.

Legal values: true / false
Default: false
Settings Tool tab: Debugging
Scope: Global

traversalBufferSize

Sets the size, in bytes, of the buffer used in traversal replication.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #traversalBufferSize: to the session's configuration to change the value for that session only. An increase in size will take effect immediately, but a decrease may not.

Legal values: any positive Integer. The actual setting must be a multiple of 8, larger than 2048. If an illegal number is entered, it will be replaced with the nearest legal number.
Default: 250000
Settings Tool tab: Server Communication
Scope: Session-specific

verbose

When `true`, GemBuilder prints messages to the Transcript when certain events occur, such as logging a session in or out, or committing or aborting a transaction. When `false`, these messages are not printed.

Legal values: true / false
Default: true
Settings Tool tab: User Interface
Scope: Global

A.3 The Settings Browser

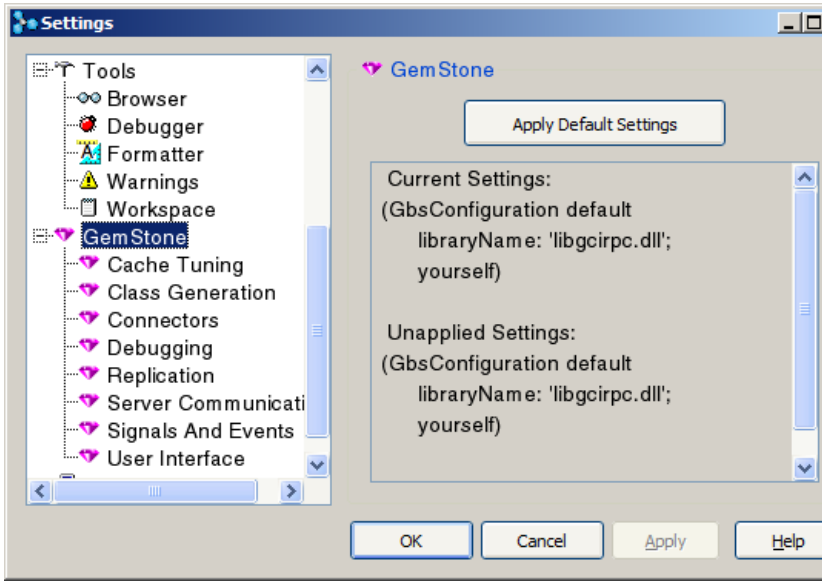
The Settings Browser makes it easy to examine and set the configuration parameters for GemBuilder. The Settings Browser is integrated with the client Smalltalk settings.

Opening the Settings Browser

To open the Settings Browser, select **Tools > Settings** from the GemStone menu, or **System > Settings** from the VisualWorks Launcher, and scroll down to see all the GemStone configuration settings categories.

The Settings Browser

Figure A.1 The Settings Browser Summary



Parameter Categorization

The Settings Browser categories the parameters under headings. Selecting each heading allows you to update a set of related configuration parameters.

Table A.2 Settings Browser Categorization

Cache Tuning	bulkLoad clientMapCapacity clientMapFinalizerPriority gcedObjBufferSize initialDirtyPoolSize serverMapLeafCapacity
Class Generation	generateClassConnectors generateServerClasses generateClientClasses
Connectors	connectorNilling connectVerification removeInvalidConnectors
Debugging	assertionChecks deprecationWarnings forwarderDebugging stubDebugging
Replication	autoMarkDirty blockReplicationEnabled blockReplicationPolicy defaultFaultPolicy faultLevelLnk faultLevelRpc freeSlotsOnStubbing
Server Communication	blockingProtocolRpc fullCompression libraryName pollForRpcResponse rpcSocketWaitTimeoutMs traversalBufferSize

Table A.2 Settings Browser Categorization

Signals And Events	eventPollingFrequency eventPriority pollForAsynchronousEvents
User Interface	alwaysUseGemCursor confirm verbose

The Settings Browser has the following buttons.

Table A.3 Buttons in the Settings Browser

Apply Default Settings	(At the highest level only) Reset all GemStone settings to their default value.
OK	Close the Settings Browser, applying all changes to the current configuration.
Cancel	Close the Settings Browser, cancelling all unapplied changes.
Apply	Apply all unapplied changes to the current configuration.
Help	Open a dialog with help for the configurations on the current view.

Appendix

B Packaging Runtime Applications

Use the following guidelines when packaging a client Smalltalk application that uses GemBuilder to access GemStone.

B.1 Prerequisites

In addition to code required by your application, the packaged image must contain the application or parcel GbsRuntime, which contains the system code modified for GemBuilder.

In order to ensure that your image initializes correctly, your application must specify GbsRuntime as a prerequisite.

Do not include the application or parcel GbsTools. These are subclasses of classes that will be deleted during the packaging process.

Names

Ensure that your image is packaged to include class pool dictionaries and instance variable names and does not remove them.

Replicating Blocks

To ensure that your application is able to replicate Smalltalk blocks in the same manner as it did in the development environment, we recommend that you include the compiler.

Defunct Stubs and Forwarders

Defunct stubs and forwarders cause problems during packaging. To avoid these problems, start with new client image as shipped from your client Smalltalk vendor.

Shared Libraries

A deployed runtime application that uses GemBuilder needs to contain all the server-specific shared libraries, as well as the as the error file `english.err` or `englis`*`xx`*`.err` (where *xx* is the release number). The Installation Guide for the GemStone server product and version you are using will provide the specific names of the library files.

If you are logging in only remote sessions, set the GemBuilder configuration parameter `libraryName` to the RPC version of the GemStone client libraries. Again, see the appropriate installation guide for the specific library names.

B.2 Packaging

Step 1. Open a new client image as shipped from your client Smalltalk vendor.

Step 2. Ensure that you have satisfied the prerequisites given above.

Step 3. Load your application code.

Step 4. Follow the packaging instructions given by your Smalltalk vendor.

Appendix

C Client Smalltalk and GemStone Smalltalk

This appendix outlines the few general and syntactical differences between the VisualWorks and GemStone Smalltalk languages.

C.1 Language Differences

GemStone's Smalltalk language is very similar to client Smalltalk in both its organization and its syntax. GemStone Smalltalk extends the Smalltalk language with classes and primitives to add multiuser features such as transaction support and persistence. The GemStone class hierarchy is extensible, and new classes can be added as required to model an application. The GemStone class hierarchy is described in the *GemStone Programming Guide*.

A quick look at the GemStone class hierarchy shows that it differs from the client Smalltalk class hierarchy in that classes for screen manipulation and the client Smalltalk programming environment don't exist, and in that the GemStone Smalltalk hierarchy contains classes for transaction control, accounting, ownership, authorization, replication, user profiles, and indexing collections.

GemStone Smalltalk also introduces constraints and optimized selection blocks.

As a Smalltalk programmer, you will feel quite at home with GemStone Smalltalk, but you should take note of the differences outlined in this appendix.

Selection Blocks

Selection blocks in GemStone Smalltalk and the use of dots for path notation have no counterparts in client Smalltalk.

```
myEmployees select: {:i | i.is.permanent}
```

Array Constructors

Array constructors do not exist in client Smalltalk. In GemStone, array constructors:

- use square brackets,
- use commas as separators, and
- can contain any valid GemStone Smalltalk expression as an element.

```
#['string one', #symbolOne, $c, 4, Object new]
```

Exception handling in VisualWorks is accomplished by sending `Block>>on:do:`. In VisualWorks the argument to `on:` is a predefined Exception class. From within the handler block, messages can be sent to the exception instance to cause the flow of control to resume, exit the `on:do:` receiver block, or restart the block.

C.2 TimeZone handling

The GemStone server, as a multi-user system, may have a number of TimeZones installed, although only one is the current TimeZone for a particular session. The instances of TimeZone include the rules governing such things as the start and end of Daylight Savings Time. GemStone server TimeZones are created based on the Zoneinfo or Olson TimeZone repository.

DateTimes internally store the time in UTC (GMT), but display themselves based on the local current TimeZone. DateTimes do reference an instance of TimeZone, but most server operations use the gem's current TimeZone.

When server DateTimes are replicated to client TimeStamps, the GbxTimeZone is used to determine the TimeStamp's correct current local time. How the GbsTimeZone is set depends on the version of VisualWorks.

In VisualWorks version 7.7 and above, which implements a similar TimeZone scheme as GemStone, the GbxTimeZone is set to the VisualWorks default TimeZone.

In VisualWorks versions prior to 7.7, the gem's current TimeZone instance is replicated to the VisualWorks client for the GbxTimeZone.

If the desired TimeZone is not the VisualWorks default, or if the GBS application changes the gem's current TimeZone after a session has logged in, GBS cannot detect this. In this case, the client application needs to send the new message

```
GbsSession >> setClientTimeZoneFromServer
```

to re-replicate a copy of the timezone.

To explicity set a specific time zone for the client, you can create the desired TimeZone on the server, and replicate it to the client, using the method

```
GbsSession >> clientTimeZone:
```

For example:

```
myGbsSession clientTimeZone:
  (mySession evaluate: 'TimeZone
      fromGemPath:''/foo/bar/America/New_York''').
```

This would be the case if the gem and client are in different time zones, and you want the time zone to be different between the gem and client (in VisualWorks versions prior to 7.7); or if the gem and client are in different time zones, and you want the timezone to be same between the gem and client (in VisualWorks version 7.7 and later).

Index

A

abort
(GbsSession) 144, 145
(GbsSessionManager) 145, 147

abort command 54

abort request from GemStone 147

abortErrLostOtRoot signal 148

abortTransaction (GbsSession) 148

addDependent: 44

adding connector to session or global list 132

addParameters (GbsSession) 37

addToCommitOrAbortReleaseLocksSet: (System) 155

addToCommitReleaseLocksSet: (System) 155

alwaysUseGemCursor configuration parameter 217, 219

application design 25–27, 82–124

argument in message to forwarder 88

array constructors in GemStone Smalltalk 238

assertionChecks configuration parameter 217, 219

assigning a migration destination 191

authorization
and migration 192

autoMarkDirty configuration parameter 217, 219

automatic class generation 84–85, 227
disabling 85

automatic mark dirty
in VisualWorks 5i 92
in VisualWorks 7.x 91

automatic transaction mode 148, 149
defined 148

B

block
callback 113
replicating 109, 220

blockingProtocolRpc configuration parameter 217, 219

blockReplicationEnabled configuration

parameter 110, 217, 220
blockReplicationPolicy configuration parameter 217, 220
Breakpoint Browser 76–77
breakpoints 70, 76
and primitive methods 75
and special methods 75
methods that cannot have 75
bulkLoad configuration parameter 217, 220
business objects 26

C

cache
space management 207
cache inventory statistics 203
callback for blocks 113
changed object notification 157, 235
changing
connector initialization 140
postconnect action 140
schema 190
class versions and 194
shared data 90, 150
choosing the locus of execution 204
circular constraints 59
class 83
connector 127, 227
connection order 128
connector, updating 84
customizing faulting 99
filing out 65
generating automatically 84–85, 227
mapping 22, 194
mapping to one with a different storage format 109
migrating instances to a new version 190
nonforwarding 126
reduced-conflict 156
structure, matching on client and server 83
updating definitions with connectors 126
versions and replication specifications 101
versions of 190
class instance variable connector 127
class variable connector 127
Classes pane
in GemStone Browser 53
`clearCommitOrAbortReleaseLocksSet` (System) 155
`clearCommitReleaseLocksSet` (System) 155
client forwarder 86
Client Libraries 32
clientMapCapacity configuration parameter 217, 220
clientMapFinalizerPriority configuration parameter 217, 221
code pane
menu for 56
collections with instance variables, replicating 114
`commit`
(GbsSession) 144
commit command 54
`commitAndReleaseLocks` (System) 155
committing
a transaction 22
performance 156
and flushing, compared 91
changes to the repository 144
compile in ST 197
compile in ST command
in GemStone Browser's Class menu 56
compiling
a class definition 60
in a runtime application 236
concurrency
modes, setting 152
optimistic control 151
pessimistic control 151, 152
concurrent transactions, managing 150
configuration parameters 215–231
alwaysUseGemCursor 217, 219
assertionChecks 217, 219
autoMarkDirty 217, 219
blockingProtocolRpc 217, 219

blockReplicationEnabled 110, 217, 220
blockReplicationPolicy 217, 220
bulkLoad 217, 220
clientMapCapacity 217, 220
clientMapFinalizerPriority 217, 221
confirm 217, 222
connectorNilling 89, 127, 217, 222
connectVerification 217, 222
defaultFaultPolicy 98, 217, 223
deprecationWarnings 217, 223
eventPollingFrequency 217, 223
eventPriority 217, 224
faultLevelLnk 217, 224
faultLevelRpc 217, 224
forwarderDebugging 217, 225
freeSlotsOnStubbing 217, 225
fullCompression 217, 225
gcedObjBufferSize 218, 226
GemStone
CONCURRENCY_MODE 152, 161
STN_GEM_ABORT_TIMEOUT 148
generateClassConnectors 85, 218, 226
generateClientClasses 84, 218, 226
generateServerClasses 84, 218, 227
initialDirtyPoolSize 218, 227
libraryName 32, 218, 228
pollForAsynchronousEvents 218, 228
pollForRpcResponse 218, 229
removeInvalidConnectors 218, 229
rpcSocketWaitTimeoutMs 218, 229
serverMapLeafCapacity 218, 230
setting and examining 215
global 215
session specific 216
stubDebugging 218, 230
traversalBufferSize 207, 218, 231
verbose 218, 231
confirm configuration parameter 217, 222
conflicts in transactions
reducing, and performance 156
write / write, and RcQueue 157
Connected command in Connector Browser 126, 140
connected objects, synchronizing 139
connector 121–140
adding to session or global list 132
class 127, 227
class versions and 86
connection order 128
forwarders and 86
update direction and 86
updating 84
class hierarchy 130
class instance variable 127
class variable 127
connecting object networks 122
connection order 128
controlling 133
creating automatically 226
creating interactively 139
creating programmatically 130
defined 83, 121
fast 129
for kernel classes 127
global 124, 137
initializing 125
introduction to 22
list of 132
name 127
nilling 89, 127, 222
postconnect action 87, 125
removing duplicates 125
removing invalid 229
removing unresolved 138
scope 124
session 137
setting postconnect action programmatically 131
setup for initial storage of data in GemStone 140
updateGS postconnect action 140
updateST postconnect action 140
updating class definitions and 126
verifying 125, 138, 222
Connector Browser 136–140
updateGS postconnect action 140

updateST postconnect action 140
connectorNilling configuration parameter 89, 127, 217, 222
connectVerification configuration parameter 217, 222
constraints
circular 59
on instance variables 59
contexts 70
controlling the size of the client Smalltalk object cache 207
converting among forwarders, stubs, replicates, and delegates 117
`copyEmpty:`, implementing for collections with instance variables 114
create access 197
create access command
in Browser's Class menu 56
create in ST 197
create in ST command
in Browser's Class menu 56, 57
creating
connector interactively 139
connector programmatically 130
forwarder 87
forwarder interactively 139
linked session 36
remote session 36
subclasses 60
CstMessengerSupport parcel 27
current session 41
setting 43
tools attached to 42

D

data
cost of managing 204
modifying shared 90, 150
storage in GemStone 140
debug command 78
debugger 69, 70, 77
debugging 70–??
forwarders 225
getting stack trace without debugger 79
stubs 230
defaultFaultPolicy configuration parameter 98, 217, 223
defining GemStone errors 186
definition 196
defunct forwarder 88
during packaging 236
defunct stub 99
during packaging 236
delegate
converting 117
Delivery and Deployment 27
dependencies between objects, managing with replication specififcations 103
dependents, session 44–48
adding 44
committing a transaction 44
removing 44
Deprecated Features 28–29
`deprecated:` 28
deprecationWarnings configuration parameter 29, 217, 223
dictionaries
pool 60
specifying for a new class 61
dirty, defined 90
`disableGbsDebugger 78`
disabling
automatic class generation 85
block replication 110
Disconnected command in Connector Browser 126, 140
domain objects 26
dumpAllProcessStacks 79

E

`enableGbsDebugger 78`
error, user-defined 186
error-handling
during file in 68
`evaluate:` 115

`evaluate:context:` 116
event, polling for 223
eventPollingFrequency configuration parameter 217, 223
eventPriority configuration parameter 217, 224
examining the internal structure of a GemStone object 70
exception-handling 183–187
`exclusiveLock:` (GbsSession) 153
`exclusiveLock:ifDenied:ifChanged:` (GbsSession) 153
`exclusiveLockAll:` (GbsSession) 153
`exclusiveLockAll:ifIncomplete:` (GbsSession) 154
execution
  in GemStone 204
  in the client Smalltalk 204
  profiling 200
  tuning 204–205
explicit stubbing of objects to reclaim space 207
extents 21

F

fast connector 129
`fault` 99
fault control
  and replicates 205
  and stubs 205
fault level
  defined 95
  performance and 205
  specifying with replication specification 98
fault policy, defined 98
faulting
  at login 95
  changes from other sessions 97
  cost of 205
  customized 105–109
  customizing a class 99
  default policy for 223
  dirty GemStone objects 205
  immediate 98
  inadequate, penalties of 206
  lazy 98
  minimizing for performance tuning 206
  when a stub receives a message 96
  while flushing, error caused by 106
faulting, defined 90
faultLevelLnk configuration parameter 217, 224
faultLevelRpc configuration parameter 217, 224
`faultToLevel:` 115
file
  writing class and method definitions to 65
file in, and error-handling 68
file out 196
file out methods 196
filing out classes and methods 65
flushing 91
  and committing, compared 91
  customized 105–109
  improving performance of 227
  of dirty replicates 204
  when 91
  while faulting, error caused by 106
flushing, defined 90
forwarder 86–89
  arguments to 88
  classes that cannot become 126
  converting 117
  creating 87
  creating interactively 139
  debugging 225
  declaring in replication specification 87
  defined 83
  defunct 88
  enforcing a return of 88
  for optimization 209
  return from 88
  sending messages to 87
  to client 86
  to server 86

when to use 86
forwarderDebugging configuration parameter 217, 225
freeSlotsOnStubbing configuration parameter 217, 225
fullCompression configuration parameter 217, 225
`fwat:` 88
`fwat:ifAbsent:` 88
`fwevaluate:` 115
`fwevaluate:context:` 116

G

GbsBuffer 105
GbsClassInstVarConnector 130
GbsClassVarConnector 130
GbsConnector 130
GbsError 236
GbsFastConnector 130
GBSM global 37
GBSM, instance of GbsSessionManager 35, 144
GbsNameConnector 130
GbsRuntime 235
GbsRuntime parcel 27
GbsServerClass 87
GbsSession 35
reference to parameters 41
GbsSessionManager 35
GbsSessionParameters 35
instance creation 36
GbsSessionParameters class
instance creation 36
GbsStackDumper 79
GbsTimeZone 238
GbsTools 235
GbsTools parcel 27
gcedObjBufferSize configuration parameter 218, 226
Gem
service name 36
signaling another Gem 158
user process 20, 21
GemBuilder
overview 21
GemBuilder tools
Breakpoint Browser 76
Classes Browser 52–57
Connector Browser 136–140
debugger 77
GemStone menu 50
overview 3, 24
Segment Tool 164–173
Session Browser 38–44
Session Parameters Editor 38
Symbol List Browser 174–177
System Workspace 57
User Account Management Tools 177–??
GemStone
documentation 5
security 162–??
GemStone Browser 52
Class List pane 55
Classes pane 53
Method Categories pane 53
Symbol List pane 53
GemStone inspector 70
GemStone Smalltalk
comparing with client Smalltalk 237
debugger 77, 78
features of 23
GemStone User List (User Account Management Tools) 177
Gem-to-Gem notifiers 158
generateClassConnectors configuration parameter 85, 218, 226
generateClientClasses configuration parameter 84, 218, 226
generateServerClasses configuration parameter 84, 218, 227
global configuration parameters 215
global connectors 124
GsInterSessionSignal 158
`gsObjImpl` 109

H

hierarchy 196

I

immediate fault policy 98
`indexableSize` 108
`indexableValueAt:` 108
`indexableValueAt:put:` method 106
`indexableValues` 107
`indexableValuesBuffer` 105, 106
inheritance
  replication specification and 100
initialDirtyPoolSize configuration parameter 218, 227
initializing
  connectors 125
  connectors programmatically 131
inspecting
  in a debugger 70
inspector 70
instance migration 190
instance variables
  constraining 59
  direct access causing stub errors 96, 208
  in Collections 114
  mapping 85, 92
    in migration 192
  mapping nonmatching names 94
  maximum number in a Class 60
  modifying while faulting 105
  modifying while flushing 107
  private 63
  suppressing replication of 93
`instancesAreForwarders` 87
`instVarMap` 93
invariant objects 59

K

kernel class
  connecting connectors for 128
  connecting instances of 127

L

lazy fault policy 98
libraryName
  setting Client Libraries 32
libraryName configuration parameter 218, 228
linked application 33
linked session 33
`listInstances:` (Repository) 191
locks 150
  logging out, effect of 154
  on objects 153
  releasing 155
  removing 154
  setting 153
  stubbing and 96
locus of execution 204
logging into GemStone
  interactively 43
  programmatically 41
logging out of GemStone
  effect on locks 154
  interactively 43
  programmatically 42
  to resynchronize application state 148
login
  faulting at 95
`login` message 41
`logout` message 43
lost OT root 148

M

main statistics 200
managing
  concurrent transactions 150
  connectors 132
  space, and cache size 207
manual mark dirty 92
manual transaction mode 149
manual, organization of 4
mapping 83
  automatic 85

class 22
class versions and 86
classes 83
classes with different storage formats 109
instance variables 92
nonmatching names 94
schema coordination 194
mark dirty 91–92
marking dirty, automatic
in VisualWorks 5i 92
in VisualWorks 7.x 91
marking dirty, manually 92
maximum number of instance variables in a class 60
messages
faulting when a stub receives 96
Method Categories pane in GemStone Browser 53
methods
breakpoints in 70
filing out 65
primitive, and breakpoints 75
protecting 163
public 64
special, and breakpoints 75
without step points 75
migration
authorization errors and 192
destination 191
ignoring 192
instance variable mapping 192
methods for
`migrate` (Object) 191
`migrateFrom:instVarMap:` (Object) 193
`migrateInstances:to:` (Object) 192
`migrateTo:` (Object) 191
of instances 190
monitoring GemStone execution 200
move 197
moving data into GemStone 140
multiprocess applications 209

N

name connector 127
name of superclass, specifying 60
`namedValueAt:` 108
`namedValueAt:put:` 106
`namedValues` 107
`namedValues:indexableValues:` 105
`namedValuesBuffer` 105, 106
network
node 36
of objects, connecting 122
notification, Gem-to-Gem 158

O

object
business 26
domain 26
repository, overview 20
object-level invariance 59
optimization
and multiprocess applications 209
and traversal buffer size 207
by explicit stubbing 207
by using forwarders 209
choosing execution platform 82
choosing the execution platform 204
controlling replication level and 95
controlling the locus of execution 204
controlling the replication level 206
cost of data management 204
explicit stubbing and 98
minimizing replication cost 92–105
preventing transient stubs 206
reduced-conflict classes and 156
statistics gathering for 200
using forwarders 86, 209
using GemStone Smalltalk for searching and sorting large objects 205
using GemStone user actions and client Smalltalk primitives 209
order in which connectors are connected 128

P

packaging runtime applications 235
parameter in message to forwarder 88
password
  GemStone 35, 37
  host 36
performance 27
  choosing execution platform 82
  choosing the execution platform and 204
  client Smalltalk primitives and 209
  controlling fault level and 206
  controlling replication level and 95
  controlling the locus of execution and 204
  cost of data management and 204
  database searching and sorting 205
  determining bottlenecks 200
  enhancing replication 92
  explicit stubbing and 98, 207
  fault levels and 205
  forwarders and 209
  GemStone Smalltalk user actions and 209
  minimizing faulting of dirty GemStone objects 206
  minimizing replication cost 92–105
  multiprocess applications and 209
  preventing transient stubs 206
  reduced-conflict classes and 156
  reducing conflict and 156
  statistics gathering for tuning 200
  traversal buffer size and 207
  using forwarders 86
  using GemStone Smalltalk for searching and sorting large objects 205
pollForAsynchronousEvents configuration parameter 218, 228
pollForRpcResponse configuration parameter 218, 229
pool dictionaries 60
pool variables 60
postconnect action 125
  changing 140
  setting programmatically 131
  updateGS 140
`postFault` 106
precedence
  of multiple replication specifications 104
  of replication mechanisms 114
`preFault` 106
prerequisites 4
preventing transient stubs 206
primitives 209
  arguments to 98
  breakpoints and 75
private
  instance variables 63
Private Classes and Methods 28
Privileges Dialog 181
profiling GemStone Smalltalk execution 200
ProfMonitor class 200
programming interface 22
protecting methods 163
public methods 64

R

RcBag 156
RcCounter 156
RcKeyValueDictionary 156
RcQueue 157
  write / write conflicts and 157
read lock messages
  `readLock:` (GbsSession) 153
  `readLock:ifDenied:ifChanged:` (GbsSession) 153
  `readLockAll:` (GbsSession) 153
  `readLockAll:ifIncomplete:` (GbsSession) 154
read operations 150
read set 150
read / write transaction conflicts 150–156
reduced-conflict classes 156–157
  performance and 156
  RcBag 156
  RcCounter 156
  RcKeyValueDictionary 156
  RcQueue 157

storage and 156
reducing the number of objects in Smalltalk 207
registering a session 37
releasing locks 155
remote session 33
remove 197
`removeDependent:` 44
`removeFromCommitOrAbortReleaseLocksSet:` (System) 155
`removeFromCommitReleaseLocksSet:` (System) 155
removeInvalidConnectors configuration parameter 218, 229
`removeLock: (GbsSession)` 154
`removeLockAll: (GbsSession)` 154
`removeLocksForSession: (GbsSession)` 154
`removeParameters` (GbsSession) 37
removing
duplicate connectors 125
locks 154
unresolved connectors 138, 229
replicate 89–115
as argument to primitive method 98
converting 117
customized faulting of 99
defined 83
fault control and 205
flushing dirty 91, 204
locking and stubbing 96
preventing stubbing 99
update direction 89
when to use 89
replicating
blocks, avoiding 113
client Smalltalk blocks 109, 220
limits of 109–114
minimizing costs of 92
precedence of various mechanisms 114
suppressing instance variables 93
replication specification 99–105
class versions and 101
declaring forwarder in 87
inheritance and 100
managing dependencies between objects with 103
precedence 104
root object for 104
specifying fault levels in 98
switching among several 101
`replicationSpecSet:` 101
repository
modifying 90, 150
overview 20
reserved selectors 65
return value from forwarder 88
root objects 122–124
in replication specifications 104
RPC Gems
using blocking protocol for 219
RPC session 33
rpcSocketWaitTimeoutMs configuration parameter 218, 229
RT_ERR_SIGNAL_ABORT signal 147
runtime applications 235

S

saving
class and method definitions 65
login information 39
schema
coordinating 194
matching, and instance variable mapping 85
modification 190
class versions and 194
scope of connectors 124
security 26, 162
protecting methods 163
Segment Id 167
Segment Tool 164, 172
changing a default segment 173
changing authorization 172
displaying segments 166
examining authorization 172

File menu 167
Group menu 170
Help menu 171
Member menu 170
Report menu 171
Segment menu 169
segments
changing authorization 172
checking authorization 172
group assignment 167
Segment Tool 164
selection blocks in GemStone 238
selector, reserved 65
serverMapLeafCapacity configuration parameter 218, 230
session 31–48
control 34
classes for 34
creating linked 36
creating remote 36
current 32, 41, 43
dependents 44–48
adding 44
committing a transaction 44
removing 44
linked 33
logging in
interactively 43
programmatically 41
logging out
interactively 43
programmatically 42
managing connectors for 133
multiple 33, 41
persistence of notify set in 157
registering with GBSM 37
remote 33
RPC 33
seeing others' changes 97
signaling between 158
supplying parameters with Session Paramters Editor 38
tools attached to current 42
Session Browser 38–44
opening 38
starting 38
session main statistics 201
Session Manager main statistics 201
session parameters 35–37
adding connectors and 135
adding new 38
See also GbsSessionParameters
Session Parameters Editor 38
session-specific configuration parameters 216
setting
Client Library 32
configuration parameters 215
locks 153
Settings Browser, VisualWorks 7.x 232
shared libraries required for runtime applications 236
shared variables 60
sharing objects
determining which 25, 82
modifications and 150
`signaledAbortAction:` (GbsSession) 148
signaling one Gem from another 158
Smalltalk
GemStone, features of 23
spawn hierarchy 196
special
methods, and breakpoints 75
selectors 65
special GBSM classes 235
SpecialGemStoneObjects dictionary 109
stack
examining in GemStone 78
getting trace without debugger 79
statistics gathering 200
step points 73
methods without 75
stepping 70
STN_GEM_ABORT_TIMEOUT GemStone configuration parameter 148
Stone
name of 35

repository monitor 20, 21
storage, and reduced-conflict classes 156
storing data in GemStone 140
stub 95–99
as argument to primitive method 98
controlling the stub level 206
converting 117
debugging 230
defined 83
defunct 99
explicit control of 207
explicit creation 98
explicit stubbing 207
fault control and 205
faulting upon message receipt 96
instance variable access and 208
preventing transient 206
replicating 99
sending messages to 95
setting instance variables to `nil` 225
stubDebugging configuration parameter 218, 230
`stubYourself` 98, 207
subclassing 60
superclass, specifying name of 60
symbol dictionaries 174
Symbol List Browser 164, 174–177
copying and pasting objects 175
Dictionaries pane 174
File menu 175
Symbol List pane in GemStone Browser 53
synchronizing
client and GemStone objects 90–92
shared objects 139
SystemRepository, segments in 166

T

TimeZone 238
tools
attached to current session 42
overview 24
transaction 141–159
aborting 145
committing 22
and performance 156
committing, and session dependents 44
managing 44, 142, 144
modes 148–150
automatic 148, 149
automatic, defined 148
manual 149
manual, defined 149
switching between 149
transactionless 142
transactionless transaction mode 142
transient object stubs, preventing 206
transitive closure 124
transparency
and access to GemStone 22
traversalBufferSize (method) 207
traversalBufferSize configuration parameter 207, 218, 231

U

`updateRequest:` 45
updating
class definitions 84
replicate 89
User Account Management Tools 164
GemStone User Dialog 179, 180
GemStone User List 177
Privileges Dialog 181
user actions 209
and primitives 209
UserClasses symbol dictionary 85
user-defined errors 186
username
GemStone 35
host 36
UserProfile
purpose 162

V

variables
    pool 60
    shared 60
verbose configuration parameter 218, 231
verifying connectors 138
versions of classes 190, 194
    connecting and 86
    mapping and 86
    replication specifications and 101
VisualWorks 7.x Settings Browser 232
VSD (visual stat display) 203

W

write lock messages
    `writeLock:` (GbsSession) 153
    `writeLock:ifDenied:ifChanged:`
        (GbsSession) 153
    `writeLockAll:` (GbsSession) 153
    `writeLockAll:ifIncomplete:`
        (GbsSession) 154
write operations 150
write set 150
write / write transaction conflicts 150–156
    RcQueue and 157